THE GOOD PARTS OF AWS



DANIEL VASSALLO & JOSH PSCHORR

The Good Parts of AWS

Daniel Vassallo, Josh Pschorr

Version 1.3, 2020-02-03

Table of Contents

Preface . . . 1
Part 1: The Good Parts . . . 3
- The Default Heuristic . . . 3
- DynamoDB . . . 5
- S3 . . . 10
- EC2 . . . 14
- EC2 Auto Scaling . . . 17
- Lambda . . . 20
- ELB . . . 24
- CloudFormation . . . 28
- Route 53 . . . 31
- SQS . . . 31
- Kinesis . . . 33

Part 2: The Bootstrap Guide . . . 35
- Starting from Scratch . . . 36
- Infrastructure as Code . . . 53
- Automatic Deployments . . . 66
- Load Balancing . . . 85
- Scaling . . . 103
- Production . . . 113
- Custom Domains . . . 129
- HTTPS . . . 137
- Network Security . . . 159

Preface

This is not your typical reference book. It doesn't cover all of AWS or all its quirks. Instead, we want to help you realize which AWS features you'd be foolish not to use. Features for which you almost never need to consider alternatives. Features that have passed the test of time by being at the backbone of most things on the internet.

Making technical choices can be overwhelming. As developers, we have to make many choices from what seems like unlimited options. We have to choose a programming language, an application framework, a database, a cloud vendor, software dependencies, and many other things. AWS alone offers about 150 services, all with their own set of options and features, sometimes even overlapping with each other. In this book, we will reveal a technique we use ourselves to help make reliable technical choices without getting paralyzed in the face of so many options. This technique will be presented in the context of AWS, but it can be generalized to any other technical decision.

This is a book by Daniel Vassallo and Josh Pschorr. Between us, we have worked with AWS for 15 years, including 11 years working inside AWS. We have worked

on all sorts of web applications, from small projects to massive web services running on thousands of servers. We have been using AWS since it was just three services without a web console, and we even got to help build a small part of AWS itself.

This is an opinionated book. We only cover topics we have significant first-hand experience with. You won't find most of the knowledge we share here in the AWS docs.

Part 1: The Good Parts

The Default Heuristic

Chasing the best tool for the job is a particularly insidious trap when it comes to making progress—especially at the beginning of a project. We consider the relentless search for the best tool to be an optimization fallacy—in the same category as any other premature optimization.

Searching for the optimal option is almost always expensive. Any belief that we can easily discover the best option by exhaustively testing each one is delusional. To make matters worse, we developers tend to enjoy tinkering with new technology and figuring out how things work, and this only amplifies the vicious cycle of such a pursuit.

Instead of searching for the best option, we recommend a technique we call the default heuristic. The premise of this heuristic is that when the cost of acquiring new information is high and the consequence of deviating from a default choice is low, sticking with the default will

likely be the optimal choice.

But what should your default choice be? A default choice is any option that gives you very high confidence that it will work. Most of the time, it is something you've used before. Something you understand well. Something that has proven itself to be a reliable way for getting things done in the space you're operating in.

Your default choice doesn't have to be the theoretical best choice. It doesn't have to be the most efficient. Or the latest and greatest. Your default choice simply needs to be a reliable option to get you to your ultimate desirable outcome. Your default choice should be very unlikely to fail you; you have to be confident that it's a very safe bet. In fact, that's the only requirement.

With this heuristic, you start making all your choices based on your defaults. You would only deviate from your defaults if you realize you absolutely have to.

When you start with little experience, you might not have a default choice for everything you want to do. In this book we're going to share our own default choices when it comes to AWS services and features. We're going to explain why some things became our defaults, and why other things we don't even bother with. We hope this

information will help you build or supplement your basket of default choices, so that when you take on your next project you will be able to make choices quickly and confidently.

DynamoDB

Amazon describes DynamoDB as a database, but it's best seen as a highly-durable data structure in the cloud. A partitioned B-tree data structure, to be precise.

DynamoDB is much more similar to a Redis than it is to a MySQL. But, unlike Redis, it is immediately consistent and highly-durable, centered around that single data structure. If you put something into DynamoDB, you'll be able to read it back immediately and, for all practical purposes, you can assume that what you have put will never get lost.

It is true that DynamoDB can replace a relational database, but only if you think you can get away with storing all your data in a primitive B-tree. If so, then DynamoDB makes a great default choice for a database.

Compared to a relational database, DynamoDB requires you to do most of the data querying yourself within your application. You can either read a single value out of DynamoDB, or you can get a contiguous range of data. But if you want to aggregate, filter, or sort, you have to do that yourself, after you receive the requested data range.

Having to do most query processing on the application side isn't just inconvenient. It also comes with performance implications. Relational databases run their queries close to the data, so if you're trying to calculate the sum total value of orders per customer, then that rollup gets done while reading the data, and only the final summary (one row per customer) gets sent over the network. However, if you were to do this with DynamoDB, you'd have to get all the customer orders (one row per order), which involves a lot more data over the network, and then you have to do the rollup in your application, which is far away from the data. This characteristic will be one of the most important aspects of determining whether DynamoDB is a viable choice for your needs.

Another factor to consider is cost. Storing 1 TB in DynamoDB costs $256/month. For comparison, storing 1 TB in S3 costs $23.55/month. Data can also be compressed much more efficiently in S3, which could make this difference even bigger. However, storage cost

is rarely a large factor when deciding whether DynamoDB is a viable option. Instead, it's generally request pricing that matters most.

By default, you should start with DynamoDB's on-demand pricing and only consider provisioned capacity as a cost optimization. On-demand costs $1.25 per million writes, and $0.25 per million reads. Now, since DynamoDB is such a simple data structure, it's often not that hard to estimate how many requests you will need. You will likely be able to inspect your application and map every logical operation to a number of DynamoDB requests. For example, you might find that serving a web page will require four DynamoDB read requests. Therefore, if you expect to serve a million pages per day, your DynamoDB requests for that action would cost $1/day.

Then, if the performance characteristics of DynamoDB are compatible with your application and the on-demand request pricing is in the ballpark of acceptability, you can consider switching to provisioned capacity. On paper, that same workload that cost $1/day to serve 1 million pages would only cost $0.14/day with provisioned capacity, which seems like a very spectacular cost reduction. However, this calculation assumes both that requests are evenly distributed over the course of the day and that there is absolutely zero capacity headroom. (You

would get throttled if there were a million and one requests in a day.) Obviously, both of these assumptions are impractical. In reality, you're going to have to provision abundant headroom in order to deal with the peak request rate, as well as to handle any general uncertainty in demand. With provisioned capacity, you will have the burden to monitor your utilization and proactively provision the necessary capacity.

In general, you will almost always want to start with on-demand pricing (no capacity management burden). Then, if your usage grows significantly, you will almost always want to consider moving to provisioned capacity (significant cost savings). However, if you believe that on-demand pricing is too expensive, then DynamoDB will very likely be too expensive, even with provisioned capacity. In that case, you may want to consider a relational database, which will have very different cost characteristics than DynamoDB.

It is important to note that, with on-demand pricing, the capacity you get is not perfectly on-demand. Behind the scenes, DynamoDB adjusts a limit on the number of reads and writes per second, and these limits change based on your usage. However, this is an opaque process and, if you want to ensure that you reserve capacity for big fluctuations in usage, you may want to consider using

provisioned capacity for peace of mind.

A final word about DynamoDB indexes. They come in two flavors: local and global. Local indexes came first in early 2013, and global indexes were added just a few months later. The only advantage of local indexes is that they're immediately consistent, but they do come with a very insidious downside. Once you create a local index on a table, the property that allows a table to keep growing indefinitely goes away. Local indexes come with the constraint that all the records that share the same partition key need to fit in 10 GB, and once that allocation gets exhausted, all writes with that partition key will start failing. Unless you know for sure that you won't ever exceed this limit, we recommend avoiding local indexes.

On the other hand, global indexes don't constrain your table size in any way, but reading from them is eventually consistent (although the delay is almost always unnoticeable). Global indexes also have one insidious downside, but for most scenarios it is much less worrisome than that of local indexes. DynamoDB has an internal queue-like system between the main table and the global index, and this queue has a fixed (but opaque) size. Therefore, if the provisioned throughput of a global index happens to be insufficient to keep up with updates on the main table, then that queue can get full. When that

happens, disaster strikes: all write operations on the main table start failing. The most problematic part of this behavior is that there's no way to monitor the state of this internal queue. So, the only way to prevent it is to monitor the throttled request count on all your global indexes, and then to react quickly to any throttling by provisioning additional capacity on the affected indexes. Nevertheless, this situation tends to only happen with highly active tables, and short bursts of throttling rarely cause this problem. Global indexes are still very useful, but keep in mind the fact that they're eventually consistent and that they can indirectly affect the main table in a very consequential manner if they happen to be underprovisioned.

S3

If you're storing data—whatever it is—S3 should be the very first thing to consider using. It is highly-durable, very easy to use and, for all practical purposes, it has infinite bandwidth and infinite storage space. It is also one of the few AWS services that requires absolutely zero capacity management.

Fundamentally, you can think of S3 as a highly-durable hash table in the cloud. The key can be any string, and the value any blob of data up to 5 TB. When you upload or download S3 objects, there's an initial delay of around 20 ms before the data gets streamed at a rate of around 90 MB/s. You can have as many parallel uploads and downloads as you want—thus, the infinite bandwidth. You can also store as many objects as you want and use as much volume as you need, without either having to provision capacity in advance or experiencing any performance degradation as the scale increases.

S3 storage costs $23.55/TB/month using the default storage class. It's almost impossible to beat S3's storage costs when you factor in the in-built redundancy.

At first, you can start with the default storage class and ignore all the other classes. Unless you're storing several terabytes in S3, it is almost never worth bothering with them. In general, you can spare yourself the trouble of understanding all the implications of the different storage classes until you really need to start saving money from S3 storage costs.

Apart from the storage classes, S3 also offers a reduced redundancy option, but this one should definitely never be used. This is a legacy feature that has been around

since 2010 (before storage classes existed), and it is currently more expensive than the default storage class, but with no benefits and lower availability (in theory).

While S3 storage costs are practically unbeatable, request pricing can become expensive in certain situations. With S3, you pay $5/million uploads and $0.40/million downloads. When your S3 usage is the result of a human operation—such as somebody uploading a file, or requesting an image from a website—this cost tends to be acceptable. Serving a million people is a big deal, and paying $5 or $0.40 for that level of scale is hardly expensive. However, when your S3 usage is driven by other computers, this can change quickly. If you're touching S3 objects at a high frequency (millions of times a day), request pricing becomes an important aspect of S3's viability for your use case.

One limitation of S3 is that you cannot append to objects. If you have something that's changing rapidly (such as a log file), you have to buffer updates on your side for a while, and then periodically flush chunks to S3 as new objects. This buffering can reduce your overall data durability, because the buffered data will typically sit in a single place without any replication. A solution for this issue is to buffer data in a durable queue, such as SQS or Kinesis streams, as we'll see later.

A word about static website hosting on S3. Unfortunately, S3 doesn't support HTTPS when used as a static website host, which is a problem. Web browsers will display a warning, and search engines will penalize you in the rankings. You could set up HTTPS using CloudFront, but it's probably much more trouble than it's worth. Nowadays, there are plenty of static website hosts outside of AWS that offer a much better hosting experience for static websites.

Finally, a note on one of S3's quirks. Bucket names are globally unique across all AWS customers and across all AWS regions. This can be quite inconvenient because, if you try to rely on a naming convention for your bucket names, you might find that someone else has already taken the name you want. A common mitigation is to always add your AWS account ID to the bucket name, which makes conflicts much less likely.

Apart from the inconvenience, the global bucket namespace also has an important security implication. If your application tries to use an S3 bucket without checking its owner, you might find yourself uploading data to someone else's bucket. Luckily, S3 has an API to check if you own the bucket, and this should always be done before interacting with an existing S3 bucket.

EC2

EC2 allows you to get a complete computer in the cloud in a matter of seconds. The nice thing about EC2 is that the computer you get will be very similar to the computer you use to develop your software. If you can run your software on your computer, you can almost certainly run it on EC2 without any changes. This is one of EC2's main advantages compared to other types of compute platforms (such as Lambda): you don't have to adapt your application to your host.

EC2 is a sophisticated service with dozens of options that you will likely never need. This is the result of the highly varied workloads and use cases serviced by EC2. Nevertheless, the defaults that EC2 comes with are good default choices, and the most consequential decision you will have to make is selecting an instance type. As of the time of writing, EC2 offers 256 different instance types, but they can be narrowed down to a few categories defined by what they're optimized for: CPU, memory, network, storage, etc., with different instance sizes for each category.

If you were building your own server, there would be an infinite number of ways to configure it, but with EC2 you

get to pick an instance type from its catalog. Sometimes this may seem inefficient, because the instance type you settle for might come with resources you don't need. But this commoditization of server types is what makes it possible for EC2 to exist as a service and to have servers available to be provisioned in a matter of seconds.

One of the most compelling features of EC2 is that you only pay for the number of seconds your instance is running. If you choose to terminate your instance because you don't need it anymore or because you want to use a different instance type, you immediately stop getting charged for the terminated instance. However, EC2 also offers you the option to commit to a long period in exchange for a price reduction. The way this has been done for many years was through reserved instances, where you make 1- or 3-year commitments on a specific instance type in exchange for a substantial price reduction. However, a recently released option called savings plans offers equivalent cost savings with some additional flexibility in switching instance types during the period under contract. With the introduction of savings plans, we don't see any reason to use reserved instances anymore.

Spot instances are another cost saving option, where instead of saving money by reserving long-term usage,

you save money by allowing EC2 to take away your instance whenever it wants. The cost savings with spot can be even more significant than with reserved instances, but of course not every use case can tolerate having compute capacity taken away from it randomly.

In general, you should think of savings plans, reserved instances, and spot instances as just cost optimization features. You can ignore them and all their implications until you need to start improving your EC2 efficiency. Remember that these are not free discounts. They come at the cost of more complicated capacity management and less optionality.

Let's end with what's probably one of the most daunting aspects of EC2—network security. It is complicated because there are many options, but once again the defaults are a very reasonable starting point. There are two important concepts that you will likely have to modify: the security group and the VPC ACL. You can think of security groups as individual firewalls for your instances, and the VPC ACL as a network firewall. With security groups you can control what goes in and out of your instances, and with the VPC ACL you can control what goes in and out of your network. In *Part 2: The Bootstrap Guide*, we'll show you how to set up a robust EC2 setup with all the necessary networking resources.

EC2 Auto Scaling

Amazon will tell you that Auto Scaling allows you to automatically add or remove EC2 instances based on the fluctuating demands of your application. This sounds great in theory, and while we've certainly seen that work successfully in a few places, it's almost never useful except in very specific situations. You will almost never need to use the auto part of Auto Scaling for the reason it exists.

Let's start by seeing how you should decide how many EC2 instances to run. You obviously need to have enough instances to meet your expected peak demand. But you probably don't want your capacity to exactly match the demand with no leeway. You will want to have some headroom too. This headroom is not waste—it will act as a safety buffer that can absorb many types of unpredictable events. For example, if an availability zone were to go down and you lost half of your instances, the headroom in the remaining instances can compensate for the lost capacity. Or if there were to be a sudden increase in demand, the same headroom will be immediately available to take it. Or if for some reason the performance of your system were to degrade abruptly (due to a software bug, a bad instance, etc.), that same headroom may help

compensate the excess load.

So, capacity headroom is a wonderful thing. You definitely need some. And if you can afford to, it's probably wise to have a lot of it. It can help you sleep well at night—sometimes in the literal sense.

The main premise of Auto Scaling is that once you decide how much headroom you want, you'll be able to make that headroom a constant size, even as the demand for your instances fluctuates. Therefore, a simpler way to look at Auto Scaling is to see it as just a cost reduction tool. Because what's wrong with having excess headroom during off-peak periods? Absolutely nothing. Except cost.

Therefore, the first question you should ask yourself is: are your EC2 costs high enough that any reduction in usage will be materially significant? As a thought experiment, consider if your EC2 bill were to go down by 30%—would that be a big deal for your business? If not, the effort and complexity of getting Auto Scaling working properly is probably not going to be worth it. You might as well just keep the extra headroom during off-peak periods and let it work for you in case of an emergency.

The other thing to consider is: does your EC2 demand vary enough for Auto Scaling to even matter? If the

fluctuations are not significant, or they are too abrupt, or they are not very smooth, Auto Scaling will almost certainly not work well for you.

Nevertheless, in some cases, Auto Scaling can deliver exactly what it says on the tin. If you run a business where your EC2 costs are a significant percentage of your expenses and your demand patterns are compatible with Auto Scaling's capabilities, then this can be a handy tool to help you improve your business margins.

Having said all that, you should still almost always use Auto Scaling if you're using EC2! Even if you only have one instance.

Auto Scaling has a few secondary features that quite frankly should have been part of EC2 itself. One of these features is a setting that allows Auto Scaling to automatically replace an instance if it becomes unhealthy. If you are already using a load balancer, you can use the same health checks for both the load balancer and Auto Scaling. You can also send health check signals using the Auto Scaling API, either directly from your instances (which isn't necessarily a reliable way to send unhealthy signals) or from something that monitors your instances from the outside.

The other nice thing that comes with Auto Scaling is the ability to simply add or remove instances just by updating the desired capacity setting. Auto Scaling becomes a launch template for your EC2 instances, and you get a dial that you can turn up or down depending on how many running instances you need. There is no faster way to add instances to your fleet than with this method.

Lambda

If EC2 is a complete computer in the cloud, Lambda is a code runner in the cloud. With EC2 you get an operating system, a file system, access to the server's hardware, etc. But with Lambda, you just upload some code and Amazon runs it for you. The beauty of Lambda is that it's the simplest way to run code in the cloud. It abstracts away everything except for a function interface, which you get to fill in with the code you want to run.

We think Lambda is great—definitely one of the good parts of AWS—as long as you treat it as the simple code runner that it is. A problem we often see is that people sometimes mistake Lambda for a general-purpose application host. Unlike EC2, it is very hard to run a

sophisticated piece of software on Lambda without making some very drastic changes to your application and accepting some significant new limitations from the platform.

Lambda is most suitable for small snippets of code that rarely change. We like to think of Lambda functions as part of the infrastructure rather than part of the application. In fact, one of our favorite uses for Lambda is to treat it as a plugin system for other AWS services.

For example, S3 doesn't come with an API to resize an image after uploading it to a bucket, but with Lambda, you can add that capability to S3. Application load balancers come with an API to respond with a fixed response for a given route, but they can't respond with an image. Lambda lets you make your load balancer do that. CloudFront can't rewrite a request URL based on request cookies (which is useful for A/B testing), but with Lambda, you can make CloudFront do that with just a little bit of code. CloudWatch doesn't support regex-based alerting on application logs, but you can add that feature with a few lines of Lambda code. Kinesis doesn't come with an API to filter records and write them to DynamoDB, but this is very easy to do with Lambda. CloudFormation's native modeling language has many limitations and, for example, it can't create and validate a

new TLS certificate from the AWS Certificate Manager. Using Lambda, you can extend the CloudFormation language to add (almost) any capability you want. And so on—you get the idea. Lambda is a great way to extend existing AWS features.

Treating Lambda as a general-purpose host for your applications is risky. It might look compelling at first—no servers to manage, no operating system to worry about, and no costs when unused—but Lambda's limitations are insidious hidden risks that typically reveal themselves once your application evolves into something bigger. Then, things start coming at you one after the other, and before you know it, you are spending most of your time trying to figure out convoluted solutions to Lambda's numerous limitations.

Some limitations will likely improve or go away over time. For example, a very annoying issue is the cold start when a function is invoked after a period of inactivity or when Lambda decides to start running your function on new backend workers. Another problem is the limit of 250 MB for your code bundle, including all your dependencies. Depending on the programming language you're using, you can find yourself quickly exhausting this limit. And the network bandwidth from Lambda functions seems to be very limited and unpredictable. These can all be

problematic, depending on your use case, but we're quite confident that these issues will improve over time.

But then there are other limitations that are inherent to the way Lambda works and which are less likely to go away. For example, you have to assume that every Lambda invocation is stateless. If you need to access some state, you have to use something like S3 or DynamoDB. While this works fine for a demo, it can quickly become prohibitively expensive in the real world. For example, handling a WebSocket connection on Lambda will likely require a read and write to DynamoDB for every exchanged packet, which can quickly result in a spectacularly large DynamoDB bill, even with modest activity.

Our rule of thumb is simple: If you have a small piece of code that will rarely need to be changed and that needs to run in response to something that happens in your AWS account, then Lambda is a very good default choice. You can even define the code in your CloudFormation template so that this piece of custom functionality truly becomes part of your infrastructure. For everything else, we advise a lot of caution.

That said, we are very optimistic about the future of serverless computing. The idea of abstracting away

everything in the stack beneath your code is a phenomenal advance in software development. However, when building software in the present, we have to assess the options available to us today, and while Lambda has its place, it is certainly not a substitute for EC2.

ELB

ELB is a load balancer service, and comes in three variants: Application (ALB), Network (NLB), and Classic. Classic is a legacy option, and remains there only because it works with very old AWS accounts, where you can still run EC2 instances outside of a VPC. For any new setup, you should choose one of the other two variants.

ALBs are proper reverse proxies that sit between the internet and your application. Every request to your application gets handled by the load balancer first. The load balancer then makes another request to your application and finally forwards the response from your application to the caller. ALBs have lots of features, and they support sophisticated routing rules, redirects, responses from Lambda functions, authentication, sticky sessions, and many other things.

On the other hand, NLBs behave like load balancers, but they work by routing network packets rather than by proxying HTTP requests. An NLB is more like a very sophisticated network router. When a client connects to a server through an NLB, the server would see the client as if it were connected to the client directly.

Both ALBs and NLBs support TLS/HTTPS, and they integrate very well with the AWS Certificate Manager. This lets you set up TLS certificates and forget about them. The certificates get renewed automatically and deployed to your load balancers without any downtime. And all certificates are free.

To have end-to-end TLS from the caller to your application, you will also have to enable TLS on your application. Otherwise the traffic from the load balancer to your application will travel unencrypted on part of the network. Unfortunately, certificates from the Certificate Manager cannot be exported, so you can't use them for your application. Instead, common practice is to create a self-signed certificate on your host and use that for your application. The load balancers do not validate the server's certificate (neither the name, nor the expiry), so in this case a self-signed certificate works fine.

The fact that ALBs and NLBs don't validate certificates

might seem concerning. However, since these load balancers run in a VPC, Amazon authenticates each network packet and guarantees that the packets go only to the hosts you configured in your load balancer. The protection from spoofing and man-in-the-middle is provided by Amazon. That said, keep in mind that by installing TLS certificates on your load balancers, you're letting Amazon become a man-in-the-middle itself. Amazon's hardware and software will be decrypting your network traffic and re-encrypting it when forwarding it to your application (if you enable TLS on your application). If you'd rather not trust Amazon with this responsibility, you must use an NLB with TCP passthrough (without enabling TLS on the load balancer). But in that case, you must keep a valid TLS certificate on your application host and deal with certificate renewals yourself.

Only NLBs support TCP passthrough, but since NLBs work on the network layer, they also lack support for many of the features found in ALBs. So, unless you need TCP passthrough, why would you ever want to use an NLB?

Well, ALBs have two main disadvantages: First, their proxy approach adds a few milliseconds to each request, so they're slightly slower than NLBs. Second, they may not

scale quickly enough to handle a big burst of traffic.

This is because an ALB behaves like a single-tenant system. AWS keeps track of your request rates and then automatically scales your ALB up or down based on the demand it sees. The exact logic of this behavior is opaque, so the only way to be assured that your ALB's elasticity meets your demands is to test it yourself. Or, if you know a certain amount of traffic is on the way, you can ask AWS (through a support ticket) to preemptively provide sufficient capacity for you.

On the other hand, NLBs behave like a multi-tenant system, and are scaled up and down in aggregate, rather than per individual load balancer. Therefore, in theory, you should never have an NLB fail to scale to your needs (unless you exhaust all of Amazon's capacity).

NLBs are also slightly less expensive than ALBs. But a single ALB can be used to handle multiple domains (via host-based routing) while an NLB cannot, so in some situations an ALB can be more cost-effective. Nevertheless, cost is unlikely to be the deciding factor when choosing between an ALB and an NLB.

Our recommendation is to consider using an NLB first, since it offers the peace of mind of not having to worry

about any obscure capacity limits. The fact that it's also faster and less expensive is a nice bonus.

But an ALB makes a great choice too, especially if you find value in any of its unique features. For the vast majority of use cases, you shouldn't run into its elasticity limits. And even if that were to happen, it should adapt on its own without your intervention (but only after some request throttling).

CloudFormation

When using AWS, you almost always want to use some CloudFormation (or a similar tool). It lets you create and update the things you have in AWS without having to click around on the console or write fragile scripts. It takes a while to get the hang of it, but the time savings pay off the initial investment almost immediately. Even for development, the ability to tear down everything cleanly and recreate your AWS setup in one click is extremely valuable.

With CloudFormation, you define your AWS resources as a YAML script (or JSON, but we find YAML to be much

easier to read and modify). Then you point CloudFormation to your AWS account, and it creates all the resources you defined. If you run the script again without making any changes, CloudFormation won't do anything (it's idempotent). If you make a change to one resource, it will change only that resource, plus any other resources that depend on the modified one (if necessary). If you change your mind about an update, you can safely tell CloudFormation to roll it back. You can also tell CloudFormation to tear down everything it created, and it will give you your AWS account back in the original state (with a few exceptions).

All of that works exceptionally well. However, a trap that people often fall into is to use CloudFormation a little bit too much! There are some things you will likely want to keep out of there. The problems arise when you modify something manually that should be under CloudFormation's control, because when you do that, you can expect unpredictable behavior. Sometimes it's okay. Sometimes it's an unrecoverable outcome with catastrophic consequences.

When you've touched something manually, and you run your CloudFormation script again, it will often try to revert your changes back to how they were. Sometimes it will manage to do so, but you wouldn't have wanted it to.

Sometimes it will try to reconcile, but become stuck in an endless loop.

Our rule of thumb is to let CloudFormation deal with all the AWS things that are either static or change very rarely; things such as VPC configurations, security groups, load balancers, deployment pipelines, and IAM roles. Other things, such as DynamoDB tables, Kinesis streams, Auto Scaling settings, and sometimes S3 buckets, are better managed elsewhere. You may want to handle some of these things directly from your application, or you could have another simple script that sets them up separately. Then there are some things that are so infrequently touched and so hard to automate that it just doesn't make sense to script them. For example, Route 53 domain registrations and hosted zones, certificate creation and validation from the Certificate Manager, and so on.

The test for whether your infrastructure-as-code is good enough is whether you feel confident that you can tear down your stack and bring it all up again in a few minutes without any mistakes. Spending an unbounded amount of time in pursuit of scripting everything is not advisable.

Route 53

Route 53 is a DNS service. It lets you translate domain names to IP addresses. There's nothing particularly special about Route 53's DNS capabilities. In fact, it has a few annoying (but mostly minor) limitations, such as the lack of support for ALIAS records (unless they point to AWS resources) and the lack of DNSSEC support. However, the reason we stick to using Route 53 is that, first of all, it's good enough, and secondly, it integrates very well with ELB. There is a significant benefit in having CloudFormation automatically set up your load balancer together with the DNS records for your custom domain. Route 53 makes this possible, whereas if you were to use a different DNS provider, you'd likely have to manage your DNS records manually.

SQS

SQS is a highly-durable queue in the cloud. You put messages on one end, and a consumer takes them out from the other side. The messages are consumed in almost first-in-first-out order, but the ordering is not

strict. The lack of strict ordering happens because your SQS queue is actually a bunch of queues behind the scenes. When you enqueue a message, it goes to a random queue, and when you poll, you also poll a random queue. In addition, duplicate messages can emerge within SQS, so your consumers should be prepared to handle this situation.

Like S3, SQS is one of the few AWS services that requires zero capacity management. There is no limit on the rate of messages enqueued or consumed, and you don't have to worry about any throttling limits. The number of messages stored in SQS (the backlog size) is also unlimited. As long as you can tolerate the lack of strict ordering and the possibility of duplicates, this property makes SQS a great default choice for dispatching asynchronous work.

If you really need strict ordering and exactly-once delivery (no duplicates), SQS has an option to enable this property by marking your queue as FIFO. But these FIFO queues come with a throughput limit of 300 messages per second, so they're only viable if you're certain that your message rate will remain well clear of that limit.

Kinesis

You can think of a Kinesis stream as a highly-durable linked list in the cloud. The use cases for Kinesis are often similar to those of SQS—you would typically use either Kinesis or SQS when you want to enqueue records for asynchronous processing. The main difference between the two services is that SQS can only have one consumer, while Kinesis can have many. Once an SQS message gets consumed, it gets deleted from the queue. But Kinesis records get added to a list in a stable order, and any number of consumers can read a copy of the stream by keeping a cursor over this never-ending list. Multiple consumers don't affect each other, and if one falls behind, it doesn't slow down the other consumers. Whenever consumers read data out of Kinesis, they will always get their records in the same order.

In addition to supporting multiple consumers, another benefit of Kinesis over SQS is that it can be a lot cheaper. For example, putting 1 KB messages in SQS at an average rate of 500 messages per second will cost you $34.56 per day. A Kinesis stream with 50% capacity headroom can handle that same volume for just $0.96 per day. So there can be about a massive difference in cost.

The reason for this cost profile is simple: Kinesis streams are optimized for sequential reads and sequential writes. Records get added to the end of a file, and reads always happen sequentially from a pointer on that file. Unlike SQS, records in a Kinesis stream don't get deleted when consumed, so it's a pure append-only data structure behind the scenes. Data simply ages out of a Kinesis stream once it exceeds its retention period, which is 24 hours by default.

On the other hand, Kinesis is not as easy to use as SQS. While with SQS you can simply enqueue as many messages as you want without having to worry about capacity or throttling limits, a Kinesis stream is made up of slices of capacity called shards, and it's up to you to figure out how many shards you need, monitor shard utilization, add shards, and figure out the best way to route records to shards so that they get approximately an equivalent amount of traffic. And unfortunately, all of this is a significant operational burden. That said, Kinesis can still be much easier to use when compared to other self-hosted technologies that provide the same streaming properties.

Part 2: The Bootstrap Guide

In this part, we will walk you through getting a basic web application running in the cloud on AWS. We will start with a blank project, and build the application and its infrastructure step by step. Each step focuses on a single aspect of the infrastructure, and we will try to explain in detail what's happening, and why.



All the source code shown in this guide is available on GitHub. Each commit represents a code checkpoint from the book, and you can find it all here: AWS Bootstrap.

When interacting with AWS, we will use both the AWS console and the AWS CLI. In addition, we will also make use of the following tools:

- GitHub as the source code repository for our application and infrastructure code.
- node.js and npm to build our application.
- git for version control.
- curl to interact with our application.

Starting from Scratch

Objective

Get a simple web application running on a single EC2 instance.

Steps

1. Write a basic "hello world" web application.
2. Manually create basic AWS infrastructure to host our application.
3. Manually install our application on an EC2 instance.

In this section we will create a tiny web application and we will get it running on an EC2 instance in an AWS account. We will start by performing all the steps manually, and we will automate them in later sections. Our application is not very interesting, and the way it will be hosted is far from ideal, but it will allow us to spend the rest of this book building a well-contained AWS setup, step by step.

Creating our application



We will need `git` and `npm` installed. If you don't already have them, the best way to install them depends on your system, so it's best to check the official guidance.

git

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

npm

https://www.npmjs.com/get-npm

Now, let's create our bare-bones application, along with a git repository to store it.

terminal

```
$ mkdir aws-bootstrap && cd aws-bootstrap
$ git init
$ npm init -y
```

Our application is going to listen for HTTP requests on port 8080 and respond with "Hello World". The entire application will be in one file, `server.js`.

server.js

```
const { hostname } = require('os');
const http = require('http');

const message = 'Hello World\n';
const port = 8080; ❶

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(message);
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname()}:${port}/`);
});
```

❶ We'll run the server on port 8080 because port numbers below 1024 require root privileges.

We can run our application directly with the `node` command.

terminal

```
$ node server.js
Server running at http://localhost:8080/
```

And we can test it with `curl` from another terminal window.

terminal

```
$ curl localhost:8080
Hello World
```

Next, let's use a process manager to monitor our application so that it automatically restarts if it crashes. To do this, we need to modify our `package.json` file.

package.json

```
{
  "name": "aws-bootstrap",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "start": "node ./node_modules/pm2/bin/pm2 start ./server.js --name
hello_aws --log ../logs/app.log ", ❷ ❸
    "stop": "node ./node_modules/pm2/bin/pm2 stop hello_aws", ❹
    "build": "echo 'Building...'" ❺
  },
  "dependencies": {
    "pm2": "^4.2.0" ❶
  }
}
```

❶ Takes a dependency on `pm2`, a node process manager.

❷ From now on, we'll use `npm start` to start our application via `pm2`.

❸ `pm2` will monitor it under the name `hello_aws` and send its stdout to `../logs/app.log`.

❹ We'll use `npm stop` to tell `pm2` to stop our application.

❺ A dummy build step. Your actual production build process goes here.

Now, we need to use `npm` to get the new dependency we added in `package.json`.

terminal

```
$ npm install
```

In `package.json` we specified that the application's logs are sent to `../logs`. Having the directory for logs outside the application's directory will be important when we deploy this with CodeDeploy, since it prevents the logs directory from being deleted with every deployment. For

now, let's create the directory manually on our local machine.

terminal

```
$ mkdir ../logs
```

And now we should be able to start our application through the process manager.

terminal

```
$ npm start

[PM2] Applying action restartProcessId on app [hello_aws](ids: [ 0 ])
[PM2] [hello_aws](0) ✓
[PM2] Process successfully started
┌────┬───────────┬──────┬──┬────────┬──────┬────────┐
│ id │ name      │ mode │ ↺ │ status │ cpu  │ memory │
├────┼───────────┼──────┼──┼────────┼──────┼────────┤
│ 0  │ hello_aws │ fork │ 0 │ online │ 0%   │ 8.7mb  │
└────┴───────────┴──────┴──┴────────┴──────┴────────┘
```

Testing via `curl` should once again show our "Hello World" message.

terminal

```
$ curl localhost:8080
Hello World
```

And with everything working, we can now commit all our changes to git.

terminal

```
$ git add server.js package.json package-lock.json
$ git commit -m "Create basic hello world web application"
```

Pushing our code to GitHub

If you don't already have a GitHub account, create one (it's free for simple projects like this). You'll also need to set up SSH access.

Next, create a new GitHub repository. We named ours `aws-bootstrap`. The repository needs to be public for now, but we'll be able to make it private once we set up CodeBuild.

Now that we have a GitHub repository, let's push our local changes to GitHub.

terminal

```
$ git remote add origin git@github.com:<username>/aws-bootstrap.git ❶
$ git push -u origin master
```

❶ Replace `<username>` with your GitHub username.

Hosting our application

If you don't already have an AWS account, you will need to create one.



You should never use your AWS root account credentials other than to create an Administrator user for your account. AWS has several best practices for managing accounts and credentials.

If you are creating a new AWS account or you don't yet have an Administrator user, use the root account to create an Administrator user now, and then use only that user.



We've chosen to provide links to the AWS console in the US *East* (*N. Virginia*) (*us-east-1*) region. Feel free to choose a region closer to your physical location. Everything described here is available in all AWS regions.

Now we're going to use the AWS console to create the minimal resources necessary to get our application running in our AWS account.

First, let's go to the EC2 instances section of the AWS console.

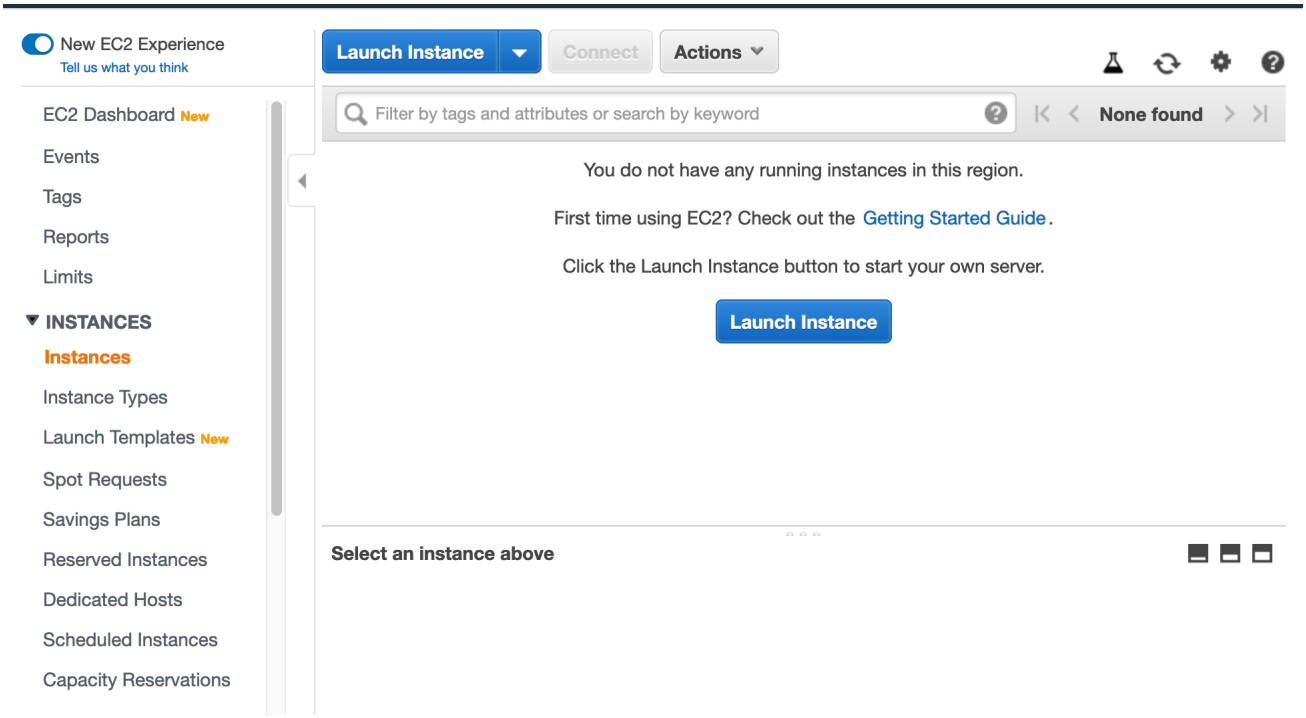


Figure 1. EC2 Instances

Hit the *Launch Instance* button, and you'll be asked to pick an AMI (Amazon Machine Image, which is basically a set of instructions for installing the operating system and some associated software). Select the default Amazon Linux 2 AMI on x86.

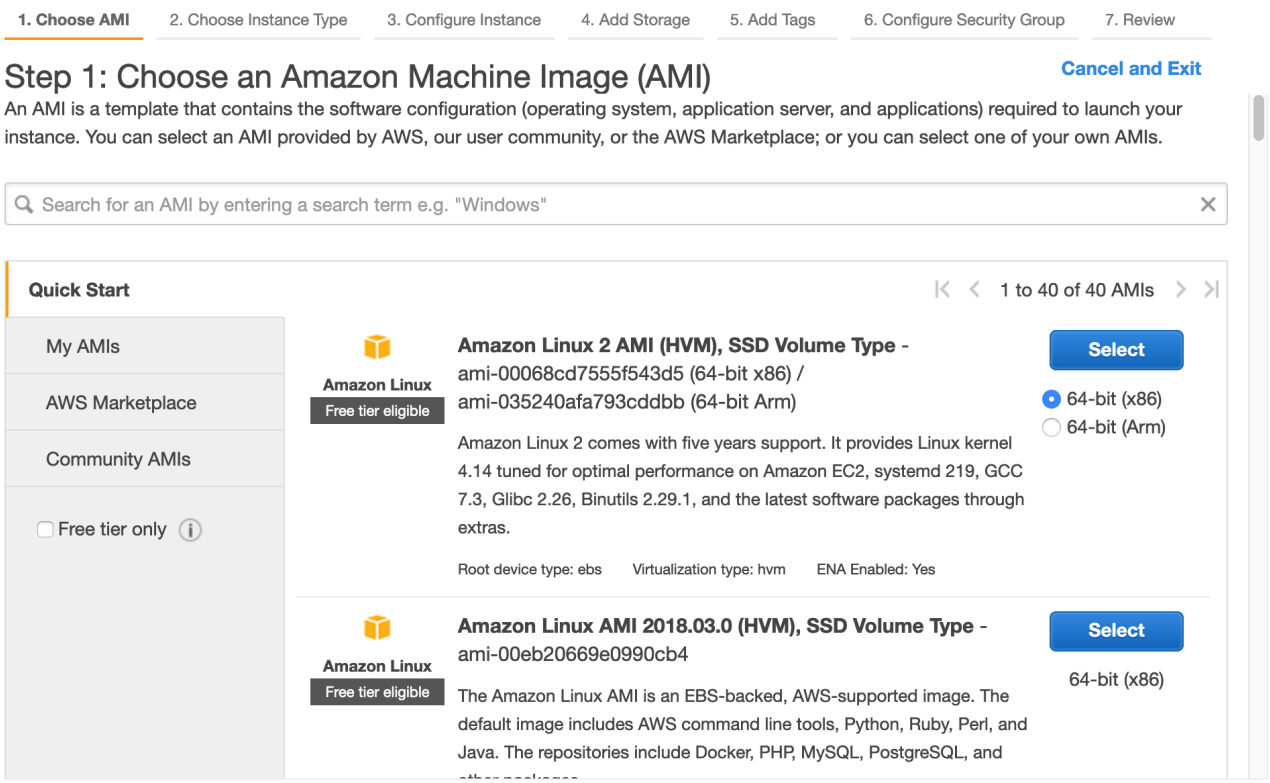


Figure 2. EC2 Instance AMI

Next, you'll be asked to choose an instance type. Again, choose the default, t2.micro, which is within the AWS free tier.

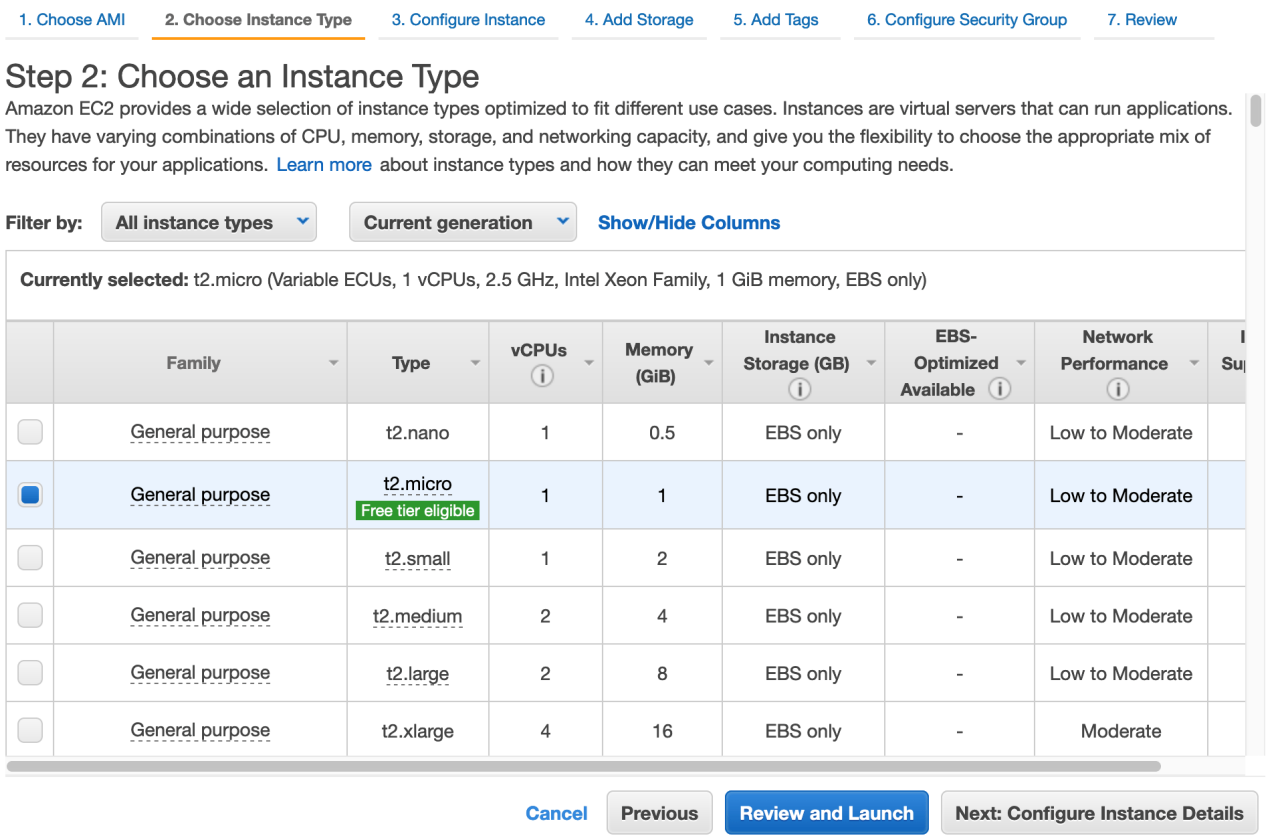


Figure 3. EC2 Instance Type

Now, we'll continue accepting the defaults until we're asked to configure the security group. Here we need to add a rule to allow traffic to port 8080, which we configured our application to listen on.

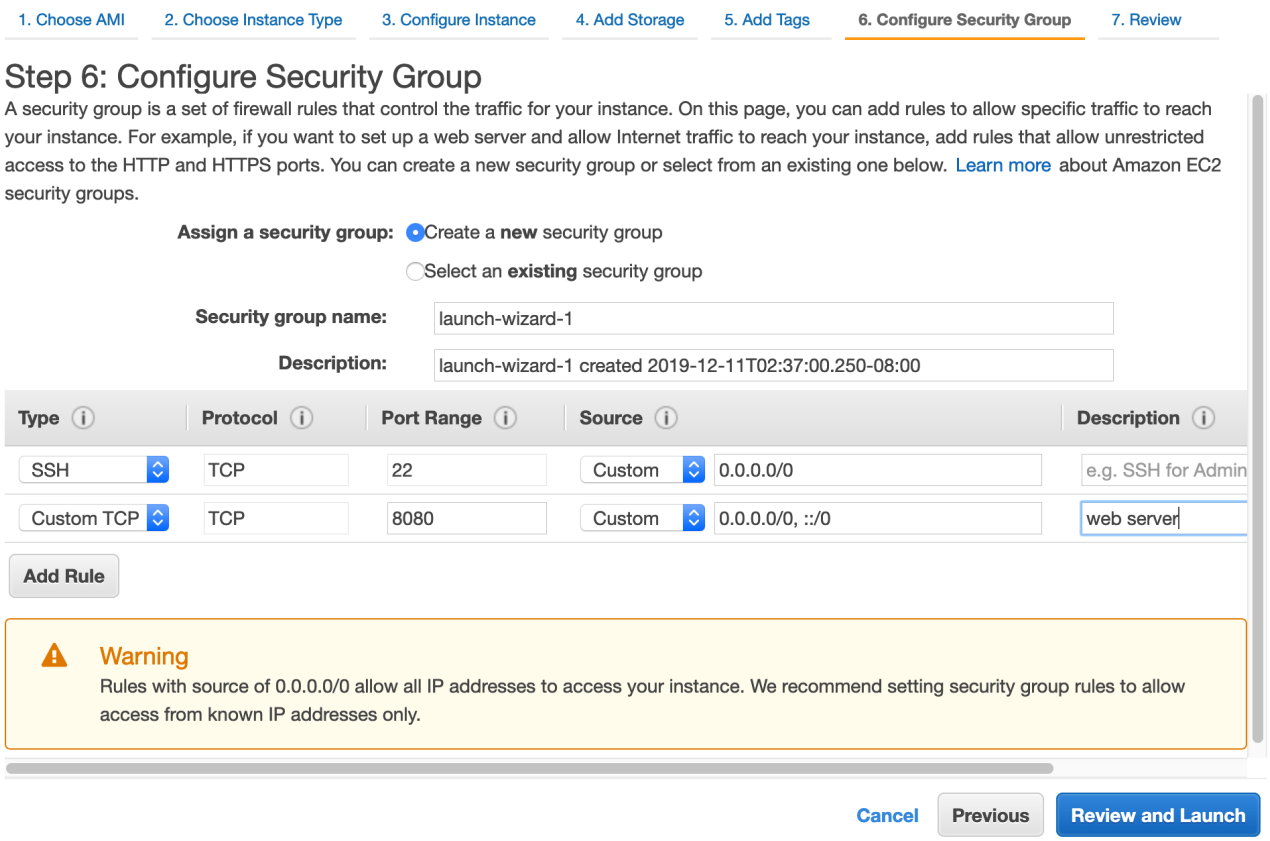


Figure 4. EC2 Security Groups

Hit the *Review and Launch* and *Launch* buttons to get our instance started. You'll be presented with a scary-looking screen saying that you won't be able to log on to the instance without a key pair. This is not entirely true, as EC2 Instance Connect provides a way to SSH into your instance without a key pair. So, select *Proceed without a key pair* and then *Launch Instance*.

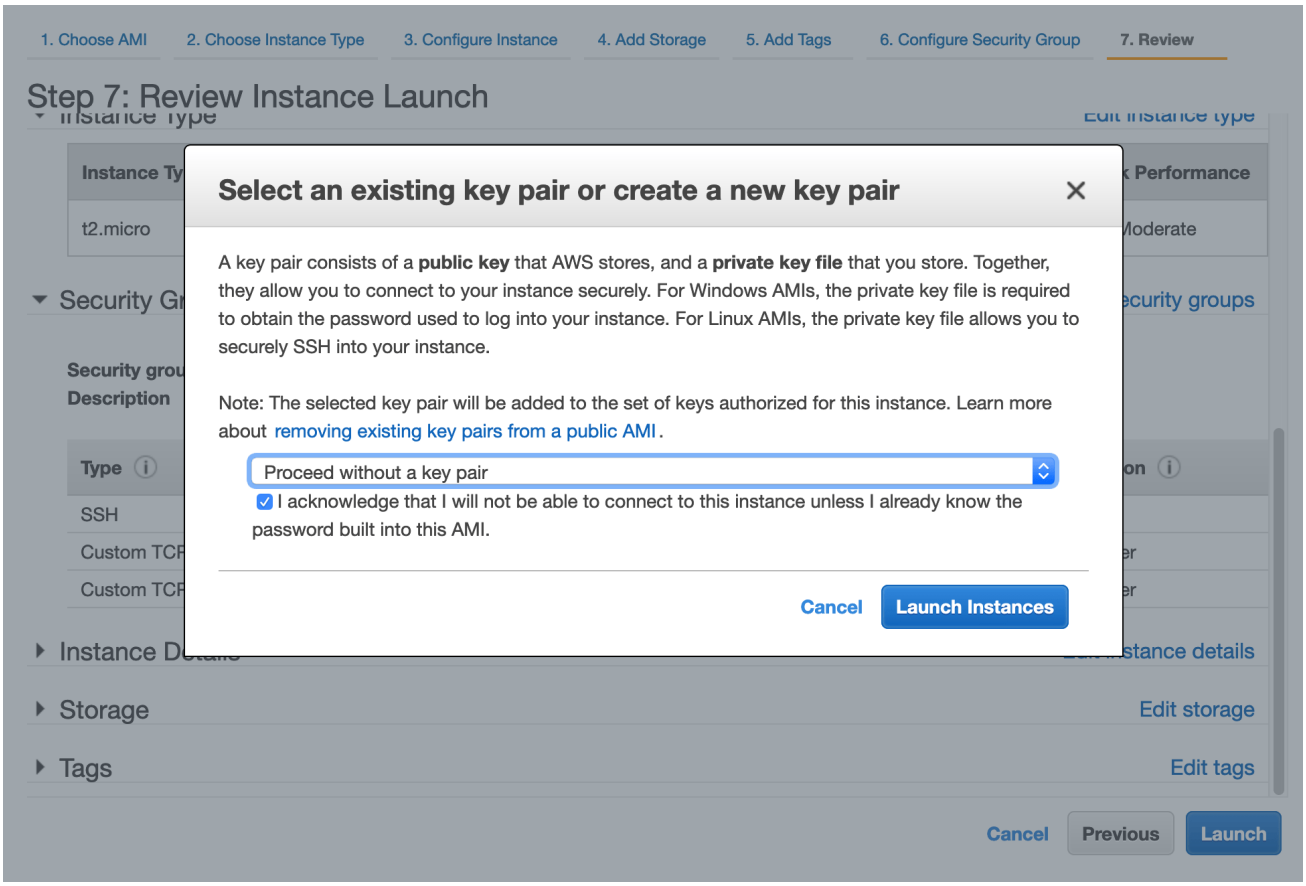


Figure 5. EC2 Instance Key Pair

Let's head to the EC2 instances page. We should see our instance running after a few seconds.

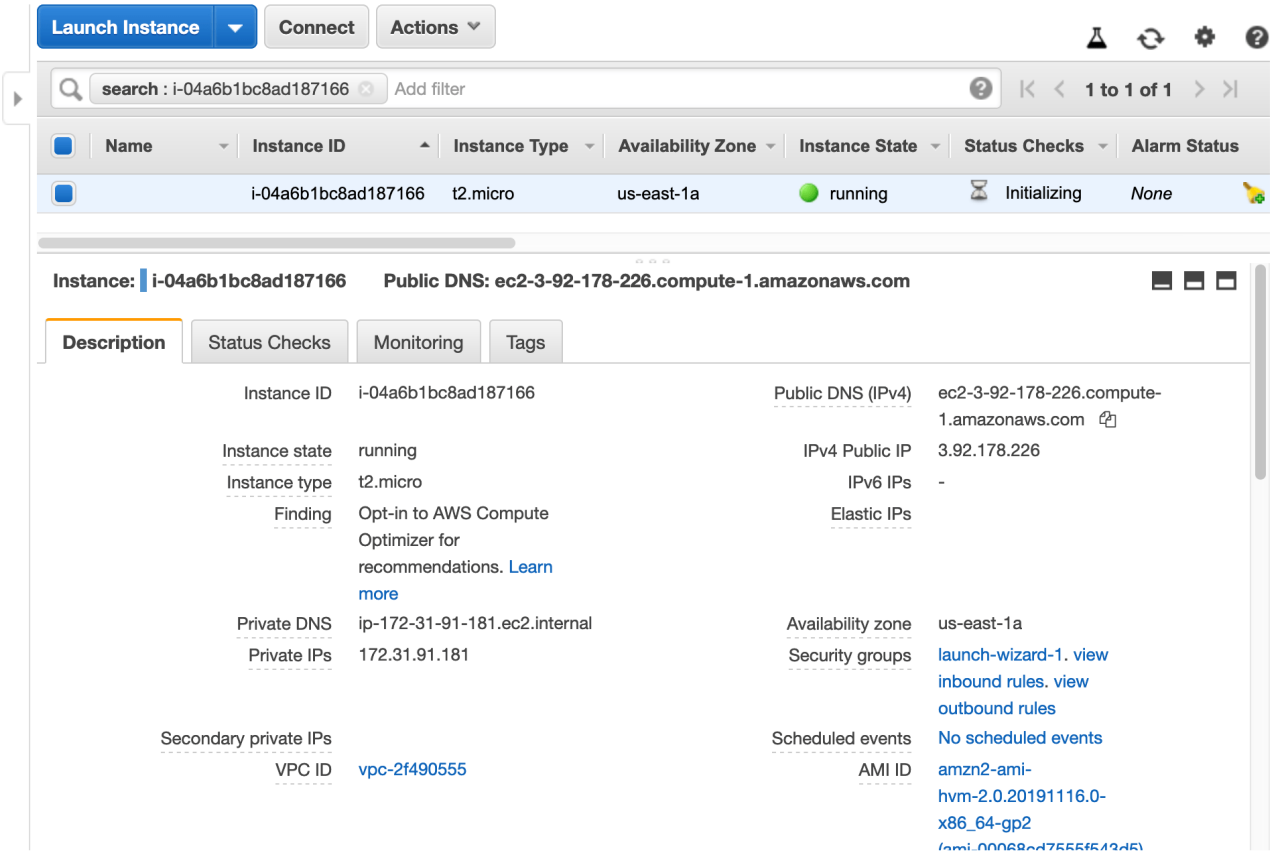


Figure 6. Running EC2 Instance

Note the Public DNS (IPv4) field shown in the console. This address will allow us to reach our instance via the internet.

Running our application

For this exercise, we're going to SSH to our instance from the browser. In the EC2 instances view, select our instance and then press the *Connect* button at the top of the page to start the connection. Then select *EC2 Instance Connect* and hit *Connect* at the bottom of the dialogue.

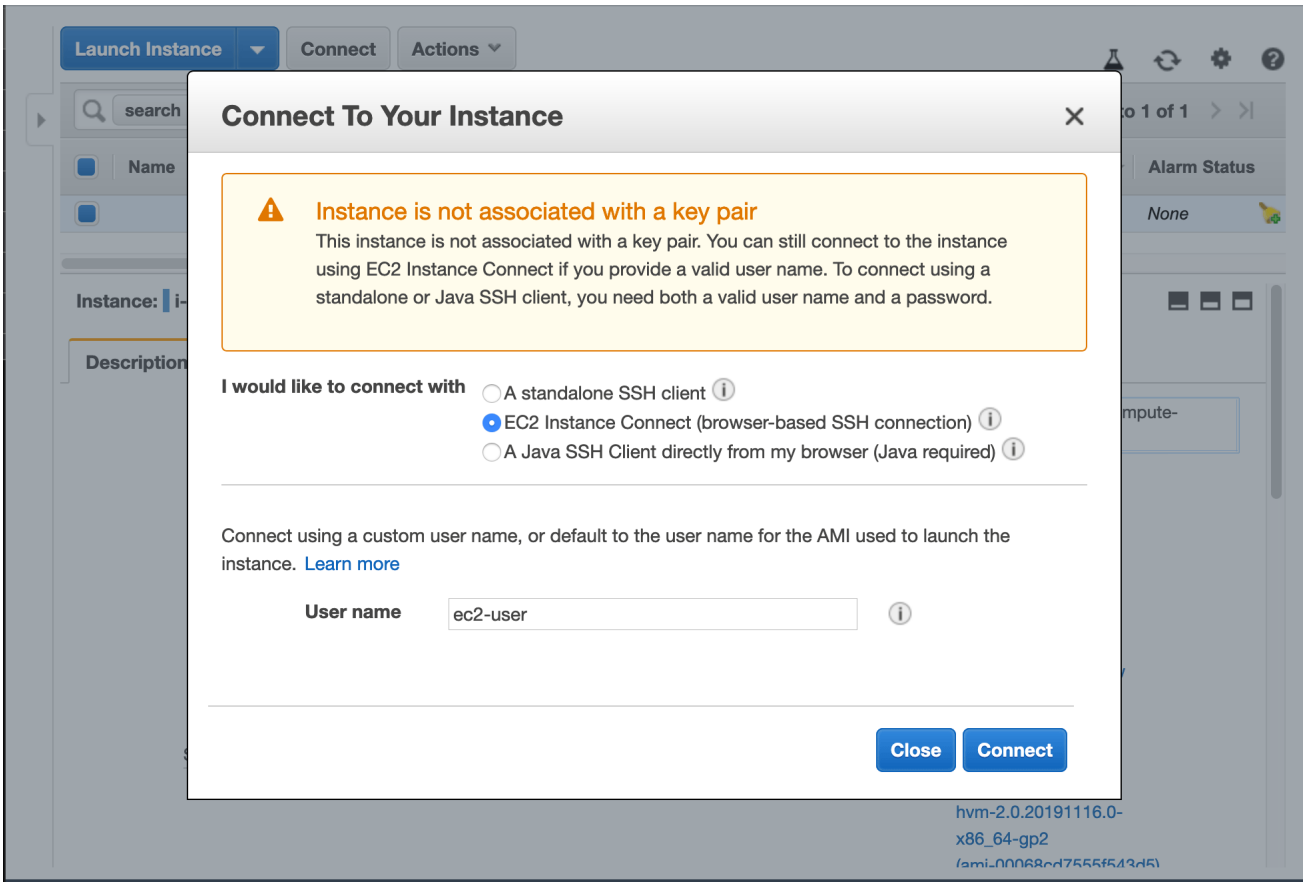


Figure 7. EC2 Connect

A new browser window will pop up with an open SSH connection to our instance.

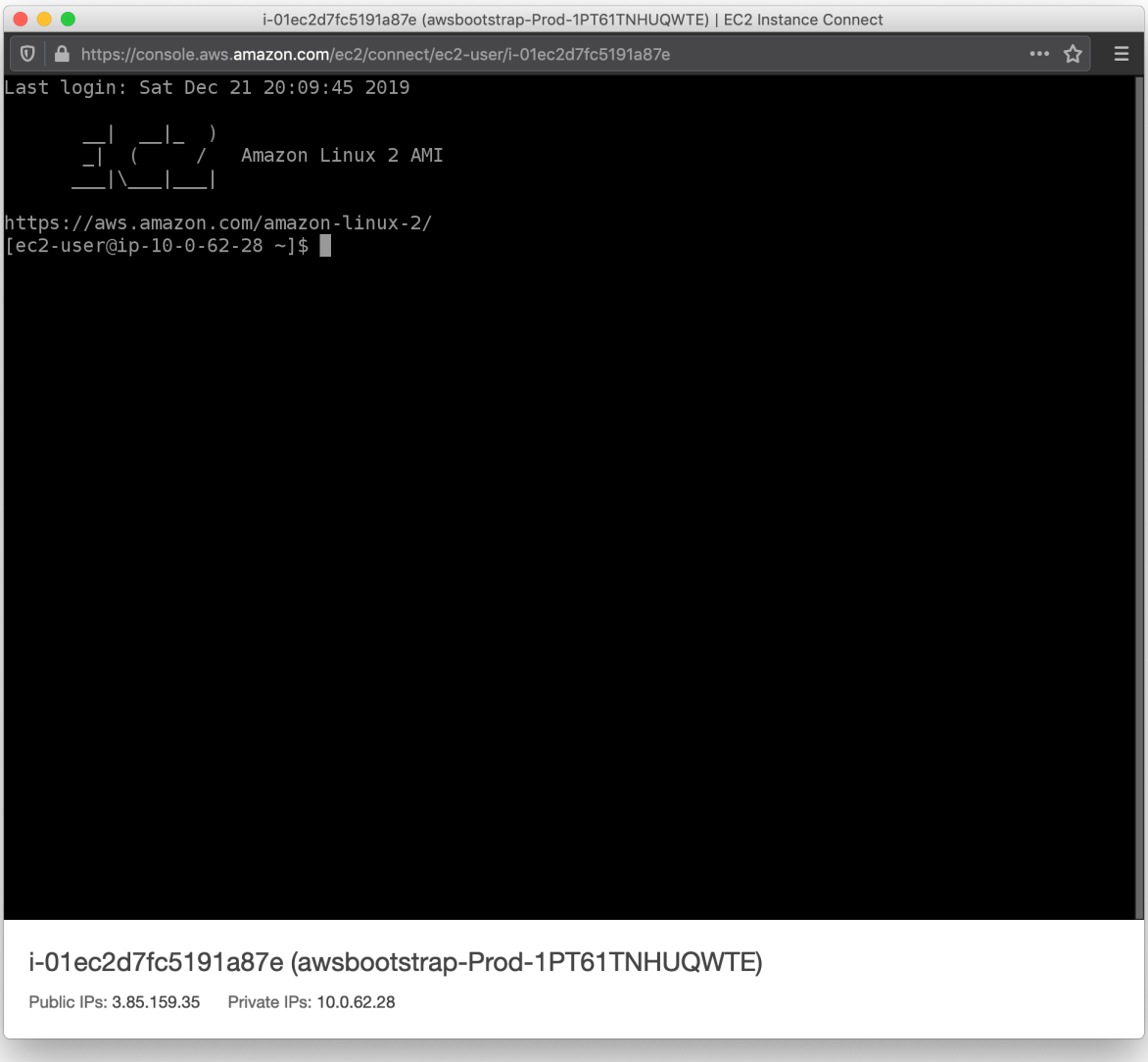


Figure 8. EC2 Connect SSH Connection

Now that we have an SSH shell open on the host, we're going to update the installed packages manually (for now). Then we will install our application and its dependencies.

terminal

```
$ sudo yum -y update ❶
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh |
bash ❷
$ . ~/.nvm/nvm.sh ❸
$ nvm install node ❹
```

❶ Updates the installed yum packages.

❷ We'll use NVM to install node.

❸ Makes sure NVM is available.

❹ Installs node via NVM.

Now that our dependencies are in place, we can proceed to install our application.

terminal

```
$ mkdir logs
$ curl -sL https://github.com/<username>/aws-bootstrap/archive/master.zip
--output master.zip  ❶ ❷
$ unzip master.zip
$ mv aws-bootstrap-master app
$ cd app
$ npm install
$ npm start
$ curl localhost:8080
Hello World
```

❶ Replace `<username>` with your GitHub user name.

❷ Here we use GitHub's archive to download the latest version of our application. This works only with public repositories.

At this point, if you copy the Public DNS (IPv4) from Figure 6, you should be able to point your web browser to `http://<public_dns>:8080` and see the "Hello World" message from our application.

Congratulations! We now have our application running in the cloud. However, our infrastructure is not yet fault-tolerant or scalable. And it's not particularly easy to

configure again, either. But we'll get there, step by step, in the following sections.



At this point, you may also want to set up billing alerts with CloudWatch to help you monitor your AWS charges and to remind you if you forget to decommission any experiments you have performed using your AWS account.

Infrastructure as Code

Objective

Recreate our infrastructure using CloudFormation.

Steps

1. Configure the AWS CLI.
2. Create a CloudFormation Stack.
3. Deploy the CloudFormation Stack.

In this section, we'll recreate the same infrastructure we set up in the previous section, but this time, we'll use CloudFormation to automate the process, instead of setting everything up manually through the AWS console.

Configuring the AWS CLI

We're going to use the AWS CLI to access AWS resources from the command line rather than the AWS console. If you don't already have it installed, follow the official directions for your system. Then, configure a profile named `awsbootstrap` using a newly generated Access Key ID and Secret Access Key, as described in the AWS

documentation.

terminal

```
$ aws configure --profile awsbootstrap
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-east-1
Default output format [None]: json
```

We can test our AWS CLI configuration by listing the EC2 instances in our account.

terminal

```
$ aws ec2 describe-instances --profile awsbootstrap
```

Now that we have the AWS CLI working, we could write a bunch of scripts that call the various AWS API commands to automate the process for setting up our infrastructure. But that would be very brittle and complicated. Luckily, there's a better way.

Infrastructure as code

Infrastructure as code is the idea of using the same processes and tools to update your infrastructure as you do for your application code. We will now start defining our infrastructure into files that can be linted, schema-checked, version controlled, and deployed without manual processes. Within AWS, the tool for this is CloudFormation.

We'll use the AWS CLI to submit infrastructure updates to CloudFormation. Although we could interact with CloudFormation directly from the AWS CLI, it is easier to write a script containing the necessary parameters. We'll call the script `deploy-infra.sh` and use it to deploy changes to our CloudFormation stack. A stack is what CloudFormation calls the collection of resources that are managed together as a unit.

deploy-infra.sh

```
#!/bin/bash

STACK_NAME=awsbootstrap ❶
REGION=us-east-1 ❷
CLI_PROFILE=awsbootstrap ❸

EC2_INSTANCE_TYPE=t2.micro ❹

# Deploy the CloudFormation template
echo -e "\n\n============ Deploying main.yml ============"
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME \
  --template-file main.yml \ ❺
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \ ❻
    EC2InstanceType=$EC2_INSTANCE_TYPE
```

❶ The stack name is the name that CloudFormation will use to refer to the group of resources it will manage.

❷ The region to deploy to.

❸ We use the `awsbootstrap` profile that we created in the previous section.

❹ An instance type in the free tier.

❺ The `main.yml` file is the CloudFormation template that we will use to

define our infrastructure.

❻ These correspond to the input parameters in the template that we'll write next.

Before we move on, let's make our helper script executable.

terminal

```
$ chmod +x deploy-infra.sh
```

Now it's time to start creating our CloudFormation template. It uses the following three top-level sections.

Parameters

These are the input parameters for the template. They give us the flexibility to change some settings without having to modify the template code.

Resources

This is the bulk of the template. Here is where we define and configure the resources that CloudFormation will manage for us.

Outputs

These are like return values for the template. We use them to make it easy to find some of the resources that CloudFormation will create for us.

We're going to name our template file `main.yml`. There will be other template files later, but they will all be referenced from here. This file will become quite large, so let's start by sketching out its high-level structure.

main.yml

```
AWSTemplateFormatVersion: 2010-09-09

Parameters:

Resources:

Outputs:
```

Next, let's fill in the input parameters to accept the instance type. The parameter names we'll use in the template will need to match the parameters we used in the `deploy-infra.sh` script. For now, we have `EC2InstanceType`. We will add other parameters throughout the book. `EC2AMI` is a bit special. We use the AWS SSM provided value that specifies the most up-to-date AMI.

main.yml

```
Parameters:
  EC2InstanceType:
    Type: String
  EC2AMI:
    Type: 'AWS::SSM::Parameter::Value<AWS::EC2::Image::Id>' ❶
    Default: '/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2'
```

❶ This is a special parameter type that allows our template to get the latest AMI without having to specify the exact version.

The first resource that we're going to define is our security group. This functions like a firewall for the EC2 instance that we'll create. We need to add a rule to allow TCP traffic to port 8080 (to reach our application) and to port 22 (for SSH access).

main.yml

```
Resources:

  SecurityGroup
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: !Sub 'Internal Security group for ${AWS::StackName}'
❶ ❷
      SecurityGroupIngress:
        - IpProtocol: tcp
          FromPort: 8080
          ToPort: 8080
          CidrIp: 0.0.0.0/0
        - IpProtocol: tcp
          FromPort: 22
          ToPort: 22
          CidrIp: 0.0.0.0/0
      Tags: ❸
        - Key: Name
          Value: !Ref AWS::StackName ❹
```

❶ `!Sub` is a CloudFormation function that performs string interpolation. Here, we interpolate the stack name into the security group description.

❷ `AWS::StackName` is a CloudFormation pseudo parameter. There are many other useful ones.

❸ Tags are great. There are many ways to use them. At the very least, it makes sense to tag most resources with the stack name if you are going to have multiple stacks in the same AWS account.

❹ `!Ref` is a CloudFormation function for referring to other resources in your stack.

The next resource we'll create is an IAM role, which our EC2 instance will use to define its permissions. At this point our application doesn't need much, as it isn't using any AWS services yet. For now, we will grant our instance role full access to AWS CloudWatch, but there are many other managed polices, which you can choose based on what permissions your application needs.

main.yml

```
Resources:
  SecurityGroup: ...

  InstanceRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          Effect: Allow
          Principal:
            Service:
              - "ec2.amazonaws.com"
          Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/CloudWatchFullAccess
      Tags:
        - Key: Name
          Value: !Ref AWS::StackName
```

Next, we'll create an instance profile to tie our IAM role to the EC2 instance that we'll create.

main.yml

```
Resources:
  SecurityGroup: ...
  InstanceRole: ...

  InstanceProfile:
    Type: "AWS::IAM::InstanceProfile"
    Properties:
      Roles:
        - Ref: InstanceRole
```

Now it's time to create our final resource, the EC2 instance itself.

main.yml

```
Resources:
  SecurityGroup: ...
  InstanceRole: ...
  InstanceProfile: ...

  Instance:
    Type: AWS::EC2::Instance
    CreationPolicy: ❶
      ResourceSignal:
        Timeout: PT15M
        Count: 1
    Metadata:
      AWS::CloudFormation::Init:
        config:
          packages: ❷
            yum:
              wget: []
              unzip: []
    Properties:
      ImageId: !Ref EC2AMI ❸
      InstanceType: !Ref EC2InstanceType ❹
      IamInstanceProfile: !Ref InstanceProfile
      Monitoring: true
      SecurityGroupIds:
        - !GetAtt SecurityGroup.GroupId ❺
      UserData:
        # ... ❻
      Tags:
        - Key: Name
          Value: !Ref AWS::StackName
```

❶ This tells CloudFormation to wait for a signal before marking the new instance as created (we'll see how in the install script).

❷ Here we define some prerequisites that CloudFormation will install on our instance (the `wget` and `unzip` utilities). We'll need them to install our application.

❸ The AMI ID that we take as a template parameter.

❹ The EC2 instance type that we take as a template parameter.

❺ `!GetAtt` is a CloudFormation function that can reference attributes from other resources.

❻ See the next code listing for how to fill in this part.

Next, let's fill in the `UserData` section for the EC2 instance. This allows us to run commands on our instance when it launches.

main.yml

```
UserData:
  Fn::Base64: !Sub |
    #!/bin/bash -xe

    # send script output to /tmp so we can debug boot failures
    exec > /tmp/userdata.log 2>&1 ❶

    # Update all packages
    yum -y update

    # Get latest cfn scripts;
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-
practices.html#cfninit
    yum install -y aws-cfn-bootstrap

    # Have CloudFormation install any files and packages from the metadata
    /opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --region
${AWS::Region} --resource Instance ❷

    cat > /tmp/install_script.sh << EOF ❸
      # START
      echo "Setting up NodeJS Environment"
      curl https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh |
```

```
bash

      # Dot source the files to ensure that variables are available within
the current shell
      . /home/ec2-user/.nvm/nvm.sh
      . /home/ec2-user/.bashrc

      # Install NVM, NPM, Node.JS
      nvm alias default v12.7.0
      nvm install v12.7.0
      nvm use v12.7.0

      # Download latest code, unzip it into /home/ec2-user/app
      wget https://github.com/<username>/aws-bootstrap/archive/master.zip ❹
      unzip master.zip
      mv aws-bootstrap-master app

      # Create log directory
      mkdir -p /home/ec2-user/app/logs

      # Run server
      cd app
      npm install
      npm start
    EOF

    chown ec2-user:ec2-user /tmp/install_script.sh && chmod a+x
/tmp/install_script.sh
    sleep 1; su - ec2-user -c "/tmp/install_script.sh" ❺

    # Signal to CloudFormation that the instance is ready
    /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} --region
${AWS::Region} --resource Instance ❻
```

❶ The output of the `UserData` script will be written to `/tmp/userdata.log`. Look there if you need to debug any launch issues.

❷ This is where the `wget` and `unzip` utilities will get installed.

❸ This script replaces the manual setup we ran in the previous section.

❹ Replace `<username>` with your GitHub username.

❺ Runs the install script as the ec2-user.

❻ Signals to CloudFormation that the instance setup is complete.

Finally, we can export the DNS name of our newly created

instance as a template output.

main.yml

```
Outputs:
  InstanceEndpoint:
    Description: The DNS name for the created instance
    Value: !Sub "http://${Instance.PublicDnsName}:8080" ❶
    Export:
      Name: InstanceEndpoint
```

❶ This creates a URL for the endpoint of the instance.

Now, let's add the following to the bottom of the `deploy-infra.sh` script, so that we'll get the URL where we can reach our application.

deploy-infra.sh

```
# If the deploy succeeded, show the DNS name of the created instance
if [ $? -eq 0 ]; then
  aws cloudformation list-exports \
    --profile awsbootstrap \
    --query "Exports[?Name=='InstanceEndpoint'].Value" ❶ ❷
fi
```

❶ The query expression uses JMESPath expressions to pull apart the output and return only the values that we're looking for.

❷ `InstanceEndpoint` is the export name we defined in the outputs section of our template.

Deploying

Now it's time to deploy our infrastructure. Let's run the `deploy-infra.sh` command. We can check the status of our stack from the CloudFormation console. The events

tab shows which resources are being created, modified, or destroyed.

When successful, the script should show us the URL for reaching our application.

terminal

```
$ ./deploy-infra.sh

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://ec2-35-174-3-173.compute-1.amazonaws.com:8080"
]
```

And now we can test that our application is up and running with `curl`.

terminal

```
$ curl ec2-35-174-3-173.compute-1.amazonaws.com:8080
Hello World
```

Now we can commit our infrastructure code to GitHub to checkpoint our progress.

terminal

```
$ git add deploy-infra.sh main.yml
$ git commit -m "Create infrastructure via CloudFormation"
$ git push
```

We now have our application running in the cloud, with its basic infrastructure managed through code. However,

if we make a change to our application, our EC2 instance won't be updated. Next, we will make our instance receive a new version of our application automatically as soon as a change is pushed to GitHub.



Only one action at a time can be in progress for a given CloudFormation stack. If you get an error that says your stack "is in [?] state and cannot be updated", then wait until the stack has finished its current update and try again.



If there is an error with the creation of your stack, you may get a message saying that your stack "is in ROLLBACK_COMPLETE state and cannot be updated." When this happens, you will not be able to deploy again. CloudFormation does this to give you a chance to inspect the error that caused the deployment to fail. Once you've addressed the issue, you'll need to delete the stack and redeploy it.

Automatic Deployments

Objective

Automatically update our application when a change gets pushed to GitHub.

Steps

1. Get GitHub credentials.
2. Install the CodeDeploy agent on our EC2 instance.
3. Create a CodePipeline.

In this section, we're going to use CodeBuild, CodeDeploy, and CodePipeline so that our application gets updated automatically as soon as we push a change to GitHub.

GitHub access token

We will need a GitHub access token to let CodeBuild pull changes from GitHub. To generate an access token, go to https://github.com/settings/tokens/new and click *Generate new token*. Give it *repo* and *admin:repo_hook* permissions, and click *Generate token*.

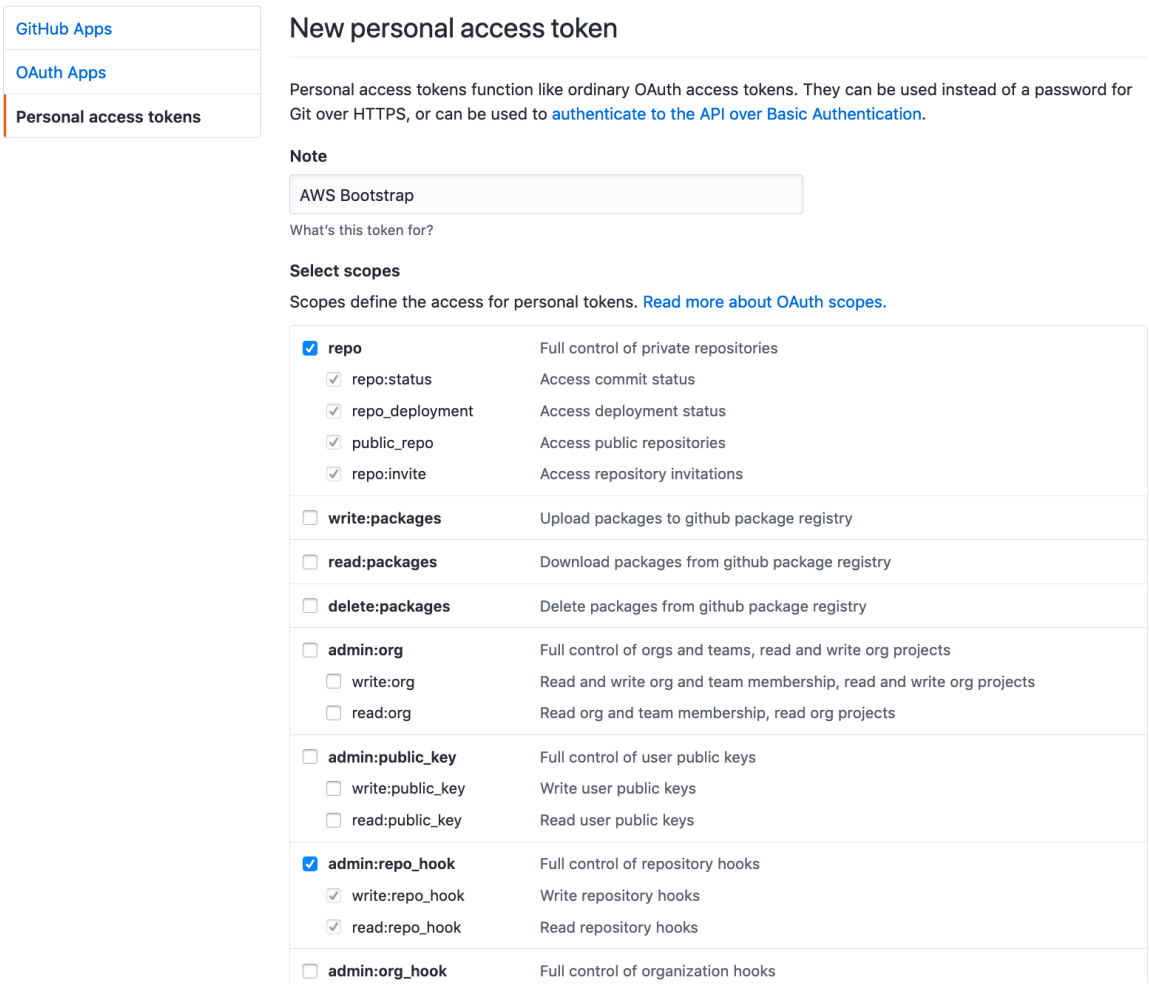


Figure 9. GitHub Access Token Generation

Tokens and passwords are sensitive information and should not be checked into source repositories. There are sophisticated ways to store them, but for now we'll put our new token in a local file that we can later read into an environment variable.

terminal

```
$ mkdir -p ~/.github
$ echo "aws-bootstrap" > ~/.github/aws-bootstrap-repo
$ echo "<username>" > ~/.github/aws-bootstrap-owner (1)
$ echo "<token>" > ~/.github/aws-bootstrap-access-token (2)
```

(1) Replace `<username>` with your GitHub username.

(2) Replace `<token>` with your GitHub access token.

S3 bucket for build artifacts

CodePipeline requires an S3 bucket to store artifacts built by CodeBuild. We chose to create this bucket outside of our main CloudFormation template because CloudFormation is unable to delete S3 buckets unless they're empty. This limitation becomes very inconvenient during development, because you would have to delete the S3 bucket manually every time you tear down your CloudFormation stack. Therefore, we like to put resources such as these in a separate CloudFormation template called `setup.yml`.

setup.yml

```
AWSTemplateFormatVersion: 2010-09-09

Parameters:
  CodePipelineBucket:
    Type: String
    Description: 'The S3 bucket for CodePipeline artifacts.'

Resources:
  CodePipelineS3Bucket:
    Type: AWS::S3::Bucket
    DeletionPolicy: Retain
    Properties:
      BucketName: !Ref CodePipelineBucket
      PublicAccessBlockConfiguration:
        BlockPublicAcls: true
        BlockPublicPolicy: true
        IgnorePublicAcls: true
        RestrictPublicBuckets: true
      BucketEncryption:
        ServerSideEncryptionConfiguration:
          - ServerSideEncryptionByDefault:
              SSEAlgorithm: AES256
```

Now let's edit our `deploy-infra.sh` script to define the

S3 bucket name for our CodePipeline.

deploy-infra.sh

```
AWS_ACCOUNT_ID=`aws sts get-caller-identity --profile awsbootstrap \ ❶
  --query "Account" --output text`
CODEPIPELINE_BUCKET="$STACK_NAME-$REGION-codepipeline-$AWS_ACCOUNT_ID" ❷
```

❶ This is a way to programmatically get the AWS account ID from the AWS CLI.

❷ S3 bucket names must be globally unique across all AWS customers. Adding our account ID to the bucket name helps prevent name conflicts.

Then we need to deploy `setup.yml` from our `deploy-infra.sh` script, just before we deploy `main.yml`.

deploy-infra.sh

```
# Deploys static resources
echo -e "\n\n=========== Deploying setup.yml ==========="
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME-setup \
  --template-file setup.yml \
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    CodePipelineBucket=$CODEPIPELINE_BUCKET
```

Start and stop scripts

Next, we need to create a couple of simple scripts to tell CodeDeploy how to start and stop our application.

start-service.sh

```
#!/bin/bash -xe
source /home/ec2-user/.bash_profile ❶
cd /home/ec2-user/app/release ❷
npm run start ❸
```

❶ Makes sure any user-specific software that we've installed (e.g., `npm` via `nvm`) is available.

❷ Changes into the working directory in which our application expects to be run.

❸ Runs the start script we put in `package.json`.

stop-service.sh

```
#!/bin/bash -xe
source /home/ec2-user/.bash_profile
[ -d "/home/ec2-user/app/release" ] && \
cd /home/ec2-user/app/release && \
npm stop
```

The build specification

Next, we need to tell CodeBuild how to build our application. To do this, CodeBuild has a specification, which we use in a file named `buildspec.yml`.

buildspec.yml

```
version: 0.2

phases:
  install:
    runtime-versions:
      nodejs: 10
  pre_build:
    commands:
      # run 'npm install' using versions in package-lock.json
      - npm ci
  build:
    commands:
      - npm run build
artifacts:
  files:
    - start-service.sh
    - stop-service.sh
    - server.js
    - package.json
    - appspec.yml
    - 'node_modules/**/*'
```

The deployment specification

Then, we need to tell CodeDeploy what to do with the build artifacts created by CodeBuild. To do this, CodeDeploy also has a specification, which we use in a file named `appspec.yml`.

appspec.yml

```
version: 0.0
os: linux
files:
  # unzip the build artifact in ~/app
  - source: /
    destination: /home/ec2-user/app/release
permissions:
  # change permissions from root to ec2-user
  - object: /home/ec2-user/app/release
    pattern: "**"
    owner: ec2-user
    group: ec2-user
hooks:
  ApplicationStart:
    # start the application
    - location: start-service.sh
      timeout: 300
      runas: ec2-user
  ApplicationStop:
    # stop the application
    - location: stop-service.sh
      timeout: 300
      runas: ec2-user
```

At this point, let's commit what we have so far to GitHub.

terminal

```
$ git add appspec.yml buildspec.yml start-service.sh stop-service.sh deploy-
infra.sh setup.yml
$ git commit -m "Add codebuild / codedeploy spec"
$ git push
```

The deployment pipeline

Now it's time to create the deployment pipeline in CloudFormation. Let's start by setting up a few environment variables in our `deploy-infra.sh` script with information about our GitHub credentials.

deploy-infra.sh

```
# Generate a personal access token with repo and admin:repo_hook
#    permissions from https://github.com/settings/tokens
GH_ACCESS_TOKEN=$(cat ~/.github/aws-bootstrap-access-token)
GH_OWNER=$(cat ~/.github/aws-bootstrap-owner)
GH_REPO=$(cat ~/.github/aws-bootstrap-repo)
GH_BRANCH=master
```

And then we can pass these variables to our `main.yml` script as parameters.

deploy-infra.sh

```
# Deploy the CloudFormation template
echo -e "\n\n============ Deploying main.yml ============"
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME \
  --template-file main.yml \
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    EC2InstanceType=$EC2_INSTANCE_TYPE \
    GitHubOwner=$GH_OWNER \
    GitHubRepo=$GH_REPO \
    GitHubBranch=$GH_BRANCH \
    GitHubPersonalAccessToken=$GH_ACCESS_TOKEN \
    CodePipelineBucket=$CODEPIPELINE_BUCKET
```

To do that, we also need to update the `Parameters` section in `main.yml` to receive the GitHub information.

main.yml

```
Parameters:
  EC2InstanceType:
    Type: String
  EC2AMI:
    Type: 'AWS::SSM::Parameter::Value<AWS::EC2::Image::Id>'
    Default: '/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2'
  CodePipelineBucket:
    Type: String
    Description: 'The S3 bucket for CodePipeline artifacts.'
  GitHubOwner:
    Type: String
    Description: 'The username of the source GitHub repo.'
  GitHubRepo:
    Type: String
    Description: 'The source GitHub repo name (without the username).'
  GitHubBranch:
    Type: String
    Default: master
    Description: 'The source GitHub branch.'
  GitHubPersonalAccessToken:
    Type: String
    NoEcho: true
    Description: 'A GitHub personal access token with "repo" and
"admin:repo_hook" permissions.'
```

Next, we need to add a new managed policy to allow our EC2 instance to access CodeDeploy.

main.yml

```
InstanceRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        Effect: Allow
        Principal:
          Service:
            - "ec2.amazonaws.com"
        Action: sts:AssumeRole
    ManagedPolicyArns:
      - arn:aws:iam::aws:policy/CloudWatchFullAccess
      - arn:aws:iam::aws:policy/service-role/AmazonEC2RoleforAWSCodeDeploy ❶
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ Allows our EC2 instance to access CodeDeploy.

We also need to create a new IAM role to allow the CodeBuild, CodeDeploy, and CodePipeline services to access our AWS resources.

main.yml

```
DeploymentRole:
  Type: AWS::IAM::Role
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        Effect: Allow
        Principal:
          Service:
            - codepipeline.amazonaws.com
            - codedeploy.amazonaws.com
            - codebuild.amazonaws.com
        Action: sts:AssumeRole
    ManagedPolicyArns:
      - arn:aws:iam::aws:policy/PowerUserAccess
```

Then we can define our CodeBuild project.

main.yml

```
BuildProject:
  Type: AWS::CodeBuild::Project
  Properties:
    Name: !Ref AWS::StackName
    ServiceRole: !GetAtt DeploymentRole.Arn
    Artifacts:
      Type: CODEPIPELINE
    Environment:
      Type: LINUX_CONTAINER
      ComputeType: BUILD_GENERAL1_SMALL
      Image: aws/codebuild/standard:2.0
    Source:
      Type: CODEPIPELINE
```

Next, we can define our CodeDeploy application. This lets CodeDeploy know that our deployment target is EC2.

main.yml

```
DeploymentApplication:
  Type: AWS::CodeDeploy::Application
  Properties:
    ApplicationName: !Ref AWS::StackName
    ComputePlatform: Server ❶
```

❶ In this case, `Server` means EC2.

To complete the CodeDeploy setup, we also need to define a deployment group. For now, we're going to have one deployment group called *Staging*. This will be our pre-production environment. We will add another deployment group for production when we get to the Production section.

main.yml

```
StagingDeploymentGroup:
  Type: AWS::CodeDeploy::DeploymentGroup
  DependsOn: Instance
  Properties:
    DeploymentGroupName: staging
    ApplicationName: !Ref DeploymentApplication
    DeploymentConfigName: CodeDeployDefault.AllAtOnce (1)
    ServiceRoleArn: !GetAtt DeploymentRole.Arn
    Ec2TagFilters: (2)
      - Key: aws:cloudformation:stack-name
        Type: KEY_AND_VALUE
        Value: !Ref AWS::StackName
```

(1) For pre-production, we can choose to deploy as fast as possible. We'll do this differently in production.

(2) These filters define how CodeDeploy will find the EC2 instances to deploy to.

And finally, we just need to define our pipeline. This comes in three stages:

1. The *Source* stage pulls the latest code from GitHub.
2. The *Build* stage builds the latest code with CodeBuild according to our `buildspec.yml` file.
3. The *Deploy* stage deploys the build artifacts from CodeBuild to the EC2 instances referenced in the deployment group, and starts the application according to our `appspec.yml` file.

main.yml

```
Pipeline:
  Type: AWS::CodePipeline::Pipeline
  Properties:
    Name: !Ref AWS::StackName
```

```
    ArtifactStore:
      Location: !Ref CodePipelineBucket
      Type: S3
    RoleArn: !GetAtt DeploymentRole.Arn
    Stages:
      - Name: Source
        Actions:
          - Name: Source
            ActionTypeId:
              Category: Source
              Owner: ThirdParty
              Version: 1
              Provider: GitHub
            OutputArtifacts:
              - Name: Source
            Configuration:
              Owner: !Ref GitHubOwner
              Repo: !Ref GitHubRepo
              Branch: !Ref GitHubBranch
              OAuthToken: !Ref GitHubPersonalAccessToken
              PollForSourceChanges: false (1)
            RunOrder: 1
      - Name: Build
        Actions:
          - Name: Build
            ActionTypeId:
              Category: Build
              Owner: AWS
              Version: 1
              Provider: CodeBuild
            InputArtifacts:
              - Name: Source
            OutputArtifacts:
              - Name: Build
            Configuration:
              ProjectName: !Ref BuildProject
            RunOrder: 1
      - Name: Staging
        Actions:
          - Name: Staging
            InputArtifacts:
              - Name: Build
            ActionTypeId:
              Category: Deploy
              Owner: AWS
              Version: 1
              Provider: CodeDeploy
            Configuration:
              ApplicationName: !Ref DeploymentApplication
              DeploymentGroupName: !Ref StagingDeploymentGroup
            RunOrder: 1
```

❶ We don't need to poll for changes because we'll set up a webhook to trigger a deployment as soon as GitHub receives a change.

Now, let's create the webhook that will trigger our pipeline as soon as a change is pushed to GitHub.

main.yml

```
PipelineWebhook:
  Type: AWS::CodePipeline::Webhook
  Properties:
    Authentication: GITHUB_HMAC
    AuthenticationConfiguration:
      SecretToken: !Ref GitHubPersonalAccessToken
    Filters:
      - JsonPath: $.ref
        MatchEquals: 'refs/heads/{Branch}'
    TargetPipeline: !Ref Pipeline
    TargetAction: Source
    Name: !Sub 'webhook-${AWS::StackName}'
    TargetPipelineVersion: !GetAtt Pipeline.Version
    RegisterWithThirdParty: true
```

We also need to make some changes to our EC2 instance to get the CodeDeploy agent installed on it.

main.yml

```
Instance:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT5M
      Count: 1
  Metadata:
    AWS::CloudFormation::Init:
      config:
        packages:
          yum:
            ruby: [] ❶
        files:
          /home/ec2-user/install: ❷
            source: !Sub "https://aws-codedeploy-
${AWS::Region}.s3.amazonaws.com/latest/install"
            mode: "000755" # executable
        commands:
          00-install-cd-agent: ❸
            command: "./install auto"
            cwd: "/home/ec2-user/"
  Properties:
    ImageId: !Ref EC2AMI
    InstanceType: !Ref EC2InstanceType
    IamInstanceProfile: !Ref InstanceProfile
    Monitoring: true
    SecurityGroupIds:
      - !GetAtt SecurityGroup.GroupId
    UserData:
      # ... ❹
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ The CodeDeploy agent requires `ruby`.

❷ Downloads the CodeDeploy agent install script to `/home/ec2-user/install` and makes it executable.

❸ Installs the CodeDeploy agent.

❹ See the next code listing for how to fill in this part.

Let's update the `UserData` section next. We need to remove the bits where we were downloading our application from GitHub because CodeDeploy will do that

for us now.

main.yml

```
UserData:
  Fn::Base64: !Sub |
    #!/bin/bash -xe

    # send script output to /tmp so we can debug boot failures
    exec > /tmp/userdata.log 2>&1

    # Update all packages
    yum -y update

    # Get latest cfn scripts;
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-
practices.html#cfninit
    yum install -y aws-cfn-bootstrap

    cat > /tmp/install_script.sh << EOF
      # START
      echo "Setting up NodeJS Environment"
      curl https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh |
bash

      # Dot source the files to ensure that variables are available within
the current shell
      . /home/ec2-user/.nvm/nvm.sh
      . /home/ec2-user/.bashrc

      # Install NVM, NPM, Node.JS
      nvm alias default v12.7.0
      nvm install v12.7.0
      nvm use v12.7.0

      # Create log directory
      mkdir -p /home/ec2-user/app/logs
    EOF

    chown ec2-user:ec2-user /tmp/install_script.sh && chmod a+x
/tmp/install_script.sh
    sleep 1; su - ec2-user -c "/tmp/install_script.sh"

    # Have CloudFormation install any files and packages from the metadata
    /opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --region
${AWS::Region} --resource Instance
    # Signal to CloudFormation that the instance is ready
    /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} --region
${AWS::Region} --resource Instance
```

And with all of that done, we can deploy our infrastructure updates. But first, we need to delete our stack from the CloudFormation console, because the changes we've made will not trigger CloudFormation to tear down our EC2 instance and start a new one. So, let's delete our stack, and recreate it by running the `deploy-infra.sh` script.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://ec2-3-93-145-152.compute-1.amazonaws.com:8080"
]
```

At this point, our EC2 instance should be up and running with the CodeDeploy agent running on it. But the CodeDeploy agent doesn't automatically deploy the application when it gets installed. For now, we can trigger the first deployment manually by hitting *Release Change* in the CodePipeline console. When we get to the Scaling section, we will have our EC2 instances deploy the

application automatically as soon as they start.

As soon as the deployment completes, we should be able to see the "Hello World" message when we visit the URL we got after running `deploy-infra.sh`.

We can now test our automatic deployments by making a change to the "Hello World" message in our application. Let's change it to "Hello Cloud" and push the changes to GitHub.

server.js

```
const message = 'Hello Cloud\n';
```

terminal

```
$ git add server.js
$ git commit -m "Change Hello World to Hello Cloud"
$ git push
```

As soon as we push the changes to GitHub, we can watch the deployment progress in the CodePipeline console. As soon as the deployment reaches the *Staging* phase, we should see "Hello Cloud" when we refresh the URL.

Our application is now getting updated automatically as soon as a change gets pushed to GitHub. And since we're now using GitHub access tokens, we can also mark our repository as private.

Let's wrap this up by pushing all our infrastructure changes to our GitHub repository.

terminal

```
$ git add setup.yml main.yml deploy-infra.sh
$ git commit -m "Add CodeDeploy/CodeBuild/CodePipeline"
$ git push
```

Load Balancing

Objective

Run our application on more than one EC2 instance.

Steps

1. Add a second EC2 instance.
2. Add an Application Load Balancer.

Currently, our application is running on a single EC2 instance. To allow our application to scale beyond the capacity of a single instance, we need to introduce a load balancer that can direct traffic to multiple instances.

Adding a second instance

We could naively add a second instance by simply duplicating the configuration we have for our existing instance. But that would create a lot of configuration duplication. Instead, we're going to pull the bulk of the EC2 configuration into an EC2 launch template, and then we'll simply reference the launch template from both instances.

We can almost copy our EC2 instance configuration into a new launch template resource as is, but there are slight differences between the two specifications. In addition, we'll also need to change the `cfn-init` and `cfn-signal` calls at the end of the `UserData` script to dynamically determine the instance ID at runtime.

main.yml

```
InstanceLaunchTemplate:
  Type: AWS::EC2::LaunchTemplate
  Metadata:
    AWS::CloudFormation::Init:
      config:
        packages:
          yum:
            ruby: []
            jq: []
        files:
          /home/ec2-user/install:
            source: !Sub "https://aws-codedeploy-
${AWS::Region}.s3.amazonaws.com/latest/install"
            mode: "000755" # executable
        commands:
          00-install-cd-agent:
            command: "./install auto"
            cwd: "/home/ec2-user/"
  Properties:
    LaunchTemplateName: !Sub 'LaunchTemplate_${AWS::StackName}'
    LaunchTemplateData:
      ImageId: !Ref EC2AMI
      InstanceType: !Ref EC2InstanceType
      IamInstanceProfile:
        Arn: !GetAtt InstanceProfile.Arn
      Monitoring:
        Enabled: true
      SecurityGroupIds:
        - !GetAtt SecurityGroup.GroupId
      UserData:
        # ... (1)
```

(1) See the next code listing for how to fill in this part.

Now let's update the `UserData` script.

main.yml

```
UserData:
  Fn::Base64: !Sub |
    #!/bin/bash -xe

    # send script output to /tmp so we can debug boot failures
    exec > /tmp/userdata.log 2>&1

    # Update all packages
    yum -y update

    # Get latest cfn scripts;
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-
practices.html#cfninit
    yum install -y aws-cfn-bootstrap

    cat > /tmp/install_script.sh << EOF
      # START
      echo "Setting up NodeJS Environment"
      curl https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh |
bash

      # Dot source the files to ensure that variables are available within
the current shell
      . /home/ec2-user/.nvm/nvm.sh
      . /home/ec2-user/.bashrc

      # Install NVM, NPM, Node.JS
      nvm alias default v12.7.0
      nvm install v12.7.0
      nvm use v12.7.0

      # Create log directory
      mkdir -p /home/ec2-user/app/logs
    EOF

    chown ec2-user:ec2-user /tmp/install_script.sh && chmod a+x
/tmp/install_script.sh
    sleep 1; su - ec2-user -c "/tmp/install_script.sh"

    # Have CloudFormation install any files and packages from the metadata
    /opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --region
${AWS::Region} --resource InstanceLaunchTemplate

    # Query the EC2 metadata service for this instance's instance-id
    export INSTANCE_ID=`curl -s http://169.254.169.254/latest/meta-
data/instance-id`❶
    # Query EC2 describeTags method and pull our the CFN Logical ID for this
instance
```

```
    export LOGICAL_ID=`aws --region ${AWS::Region} ec2 describe-tags \ ❷ ❸
      --filters "Name=resource-id,Values=${!INSTANCE_ID}" \
                "Name=key,Values=aws:cloudformation:logical-id" \
      | jq -r ".Tags[0].Value"`

    # Signal to CloudFormation that the instance is ready
    /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} --region
${AWS::Region} --resource ${!LOGICAL_ID}
```

❶ We're using the Instance Metadata service to get the instance id.

❷ Here, we're getting the tags associated with this instance. The `aws:cloudformation:logical-id` tag is automatically attached by CloudFormation. Its value is what we pass to `cfn-signal` to signal a successful launch.

❸ Note the usage of `${!INSTANCE_ID}`. Since this is inside a CloudFormation `!Sub`, if we used `${INSTANCE_ID}`, CloudFormation would have tried to do the substitution itself. Adding the `!` tells CloudFormation to rewrite it for `bash` to interpret.

Now, we can change our instance resource to reference the new launch template.

main.yml

```
Instance:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT5M
      Count: 1
  Properties:
    LaunchTemplate:
      LaunchTemplateId: !Ref InstanceLaunchTemplate
      Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber ❶
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ Each time we update our launch template, it will get a new version number. We always want to use the latest.

Adding a second instance is now as easy as creating a new instance resource that references the same launch template.

main.yml

```
Instance2:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT5M
      Count: 1
  Properties:
    LaunchTemplate:
      LaunchTemplateId: !Ref InstanceLaunchTemplate
      Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

We also need to add an inline IAM policy to allow the `UserData` script to access the EC2 `DescribeTags` API. Let's modify the `InstanceRole` resource to add it.

main.yml

```
InstanceRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        Effect: Allow
        Principal:
          Service:
            - "ec2.amazonaws.com"
        Action: sts:AssumeRole
    ManagedPolicyArns:
      - arn:aws:iam::aws:policy/CloudWatchFullAccess
      - arn:aws:iam::aws:policy/service-role/AmazonEC2RoleforAWSCodeDeploy
    Policies:
      - PolicyName: ec2DescribeTags ❶
        PolicyDocument:
          Version: 2012-10-17
          Statement:
            - Effect: Allow
              Action: 'ec2:DescribeTags'
              Resource: '*'
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ This policy allows our instance to query EC2 for tags.

Now, we can change the output of our CloudFormation template to return a URL for both instances.

main.yml

```
Outputs:
  InstanceEndpoint1:
    Description: The DNS name for the created instance
    Value: !Sub "http://${Instance.PublicDnsName}:8080"
    Export:
      Name: InstanceEndpoint1

  InstanceEndpoint2:
    Description: The DNS name for the created instance
    Value: !Sub "http://${Instance2.PublicDnsName}:8080"
    Export:
      Name: InstanceEndpoint2
```

Finally, let's change our `deploy-infra.sh` script to give us these URLs.

deploy-infra.sh

```
# If the deploy succeeded, show the DNS name of the created instance
if [ $? -eq 0 ]; then
  aws cloudformation list-exports \
    --profile awsbootstrap \
    --query "Exports[?starts_with(Name,'InstanceEndpoint')].Value"
fi
```

If we run the `deploy-infra.sh` script now, we should see a pair of URLs when the deployment finishes.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://ec2-52-91-223-254.compute-1.amazonaws.com:8080",
    "http://ec2-3-93-145-152.compute-1.amazonaws.com:8080"
]
```

Our old instance should have been terminated, and two new instances should have started in its place.

At this point, let's also checkpoint our progress into

GitHub.

terminal

```
$ git add main.yml deploy-infra.sh
$ git commit -m "Add a second instance"
$ git push
```

Next, let's modify our application a little bit to allow us to see how our requests get routed. We can do this by simply including the hostname in the response message.

server.js

```
const { hostname } = require('os');
const http = require('http');
const message = `Hello World from ${hostname()}\n`; ❶
const port = 8080;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(message);
});
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname()}:${port}/`);
});
```

❶ Changes the message to include the hostname.

Let's push this change to GitHub and wait for the deployment to finish.

terminal

```
$ git add server.js
$ git commit -m "Add hostname to hello world message"
$ git push
```

We can follow the deployment progress from the

CodePipeline console. Once the deployment is complete, we can verify our change by making a request to both URLs.

terminal

```
$ curl http://ec2-52-91-223-254.compute-1.amazonaws.com:8080
Hello World from ip-10-0-113-245.ec2.internal
$ curl http://ec2-3-93-145-152.compute-1.amazonaws.com:8080
Hello World from ip-10-0-61-251.ec2.internal
```



The hostname that the server is printing is not the same as the *Public* DNS assigned to the instance. It is actually a private domain name that EC2 assigns to each instance. You can find this private domain in the EC2 console under the *Private* DNS field.

Adding the load balancer

Now we're going to add a load balancer to have a single endpoint for our application. Until now, we have been relying on EC2's default network configuration, but by adding a load balancer we're going to have to explicitly define a VPC setup in our CloudFormation template.

We're also going to make sure that our two instances will be running in separate availability zones. This is a fundamental requirement for ensuring high availability when using EC2. We recommend James Hamilton's video Failures at Scale and How to Ignore Them from 2012 as a good introduction to how AWS thinks about availability

zones and high availability.

The VPC and subnets will require IP address allocations, for which we need to use CIDR (Classless inter-domain routing) notation. A full discussion of CIDR concepts is beyond the scope of this book, but AWS has some documentation about this topic in VPC sizing and VPC design. For our purposes, we will be allocating our VPC addresses from one of the RFC 1918 private address ranges: `10.0.0.0 - 10.255.255.255`.

The CIDR notation for that whole address range is `10.0.0.0/8`. VPC allows us to allocate up to half of that range with a notation of `10.0.0.0/16`, which corresponds to 65,536 unique IP addresses. We'll split it evenly into 4 subnets of 16,382 addresses each: `10.0.0.0/18`, `10.0.64.0/18`, `10.0.128.0/18`, and `10.0.192.0/18`. We're going to use two blocks of addresses right now, and the other two when we get to the topic of network security.

The following might look very complicated, but it's mostly boilerplate configuration. You will rarely need to change any of these settings.

So, let's start by adding our VPC and two subnets to our CloudFormation template.

main.yml

```
VPC:
  Type: AWS::EC2::VPC
  Properties:
    CidrBlock: 10.0.0.0/16
    EnableDnsSupport: true
    EnableDnsHostnames: true
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName

SubnetAZ1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref VPC
    AvailabilityZone: !Select [ 0, !GetAZs '' ]  ❶ ❷
    CidrBlock: 10.0.0.0/18
    MapPublicIpOnLaunch: true ❸
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 0, !GetAZs '' ]

SubnetAZ2:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref VPC
    AvailabilityZone: !Select [ 1, !GetAZs '' ] ❶ ❷
    CidrBlock: 10.0.64.0/18
    MapPublicIpOnLaunch: true ❸
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 1, !GetAZs '' ]
```

❶ `!GetAZs` is a CloudFormation function that returns an array of the available availability zones.

❷ `!Select` is a CloudFormation function that pulls an object out of an array by its index.

❸ We'll need a public IP in order to SSH to the instances. We'll lock this down when we work on network security later.

We also need to add an internet gateway now that we're

no longer using the default one. The internet gateway makes it possible for our hosts to route network traffic to and from the internet.

main.yml

```
InternetGateway:
  Type: AWS::EC2::InternetGateway
  Properties:
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName

InternetGatewayAttachment:
  Type: AWS::EC2::VPCGatewayAttachment
  Properties:
    InternetGatewayId: !Ref InternetGateway
    VpcId: !Ref VPC
```

We also need to set up a routing table for our subnets to tell them to use our internet gateway.

main.yml

```
RouteTable:
  Type: AWS::EC2::RouteTable
  Properties:
    VpcId: !Ref VPC
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName

DefaultPublicRoute:
  Type: AWS::EC2::Route
  DependsOn: InternetGatewayAttachment
  Properties:
    RouteTableId: !Ref RouteTable
    DestinationCidrBlock: 0.0.0.0/0
    GatewayId: !Ref InternetGateway

SubnetRouteTableAssociationAZ1:
  Type: AWS::EC2::SubnetRouteTableAssociation
  Properties:
    RouteTableId: !Ref RouteTable
    SubnetId: !Ref SubnetAZ1

SubnetRouteTableAssociationAZ2:
  Type: AWS::EC2::SubnetRouteTableAssociation
  Properties:
    RouteTableId: !Ref RouteTable
    SubnetId: !Ref SubnetAZ2
```

Now it's time to create the load balancer itself. The load balancer will exist in both of our subnets.

main.yml

```
LoadBalancer:
  Type: AWS::ElasticLoadBalancingV2::LoadBalancer
  Properties:
    Type: application
    Scheme: internet-facing
    SecurityGroups:
      - !GetAtt SecurityGroup.GroupId
    Subnets:
      - !Ref SubnetAZ1
      - !Ref SubnetAZ2
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

Then, let's configure our load balancer to listen for HTTP traffic on port 80, and forward that traffic to a target group named `LoadBalancerTargetGroup`.

main.yml

```
LoadBalancerListener:
  Type: AWS::ElasticLoadBalancingV2::Listener
  Properties:
    DefaultActions:
      - Type: forward
        TargetGroupArn: !Ref LoadBalancerTargetGroup
    LoadBalancerArn: !Ref LoadBalancer
    Port: 80
    Protocol: HTTP
```

Now we can create the target group that references our two EC2 instances and the HTTP port they're listening on.

main.yml

```
LoadBalancerTargetGroup:
  Type: AWS::ElasticLoadBalancingV2::TargetGroup
  Properties:
    TargetType: instance
    Port: 8080
    Protocol: HTTP
    VpcId: !Ref VPC
    HealthCheckEnabled: true
    HealthCheckProtocol: HTTP
    Targets:
      - Id: !Ref Instance
      - Id: !Ref Instance2
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

We will place each instance in one of the subnets we created by adding a `SubnetId` property to each. This will also place each instance into the availability zone

specified by the subnet.

main.yml

```
Instance:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT5M
      Count: 1
  Properties:
    SubnetId: !Ref SubnetAZ1 ❶
    LaunchTemplate:
      LaunchTemplateId: !Ref InstanceLaunchTemplate
      Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName

Instance2:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT5M
      Count: 1
  Properties:
    SubnetId: !Ref SubnetAZ2 ❶
    LaunchTemplate:
      LaunchTemplateId: !Ref InstanceLaunchTemplate
      Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ Puts `Instance` in `SubnetAZ1` and `Instance2` in `SubnetAZ2`.

Now let's return to our security group resource. We'll need to add a reference to the VPC we created. We also need to open port 80, as that's what our load balancer is listening on.

main.yml

```
SecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId: !Ref VPC ❶
    GroupDescription:
      !Sub 'Internal Security group for ${AWS::StackName}'
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 8080
        ToPort: 8080
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp ❷
        FromPort: 80
        ToPort: 80
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: 0.0.0.0/0
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ References our VPC.

❷ Adds a new rule to allow internet traffic to port 80.

Finally, let's wrap up these changes by making our CloudFormation template output the domain name of the load balancer instead of the individual EC2 instances.

main.yml

```
LBEndpoint:
  Description: The DNS name for the LB
  Value: !Sub "http://${LoadBalancer.DNSName}:80"
  Export:
    Name: LBEndpoint
```

And let's also modify the `deploy-infra.sh` script to give us the load balancer's endpoint.

deploy-infra.sh

```
# If the deploy succeeded, show the DNS name of the created instance
if [ $? -eq 0 ]; then
  aws cloudformation list-exports \
    --profile awsbootstrap \
    --query "Exports[?ends_with(Name,'LBEndpoint')].Value" ❶
fi
```

❶ Using `ends_with` here is not necessary right now, but will be useful when we get to the Production section.

Now we're ready to deploy our new infrastructure.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://awsbo-LoadB-15405MBPT66BQ-476189870.us-east-
1.elb.amazonaws.com:80"
]
```

If the instances needed to be recreated, then our application won't be running on them automatically. We can trigger the deployment manually by hitting *Release Change* in the CodePipeline console.

Once the deployment is complete, we should be able to

reach our application through the load balancer endpoint. If we try to make several requests to this endpoint, we should be able to see the load balancer in action, where we get a different message depending on which of our two instances responded to the request.

terminal

```
$  for run in {1..10}; do curl -s http://awsbo-LoadB-15405MBPT66BQ-
476189870.us-east-1.elb.amazonaws.com:80; done | sort | uniq -c
   4 Hello World from ip-10-0-113-245.ec2.internal
   6 Hello World from ip-10-0-61-251.ec2.internal
```

Now it's time to push all our infrastructure changes to GitHub and move to the next section.

terminal

```
$ git add main.yml deploy-infra.sh
$ git commit -m "Add ALB Load Balancer"
$ git push
```

Scaling

Objective

Replace explicit EC2 instances with Auto Scaling.

Steps

1. Add an Auto Scaling Group.
2. Remove `Instance` and `Instance2`.

Thus far, we have created our two EC2 instances explicitly. Doing this means that we need to update the CloudFormation template and do an infrastructure deployment just to add, remove, or replace an instance.

In this section, we'll replace our two EC2 instances with an auto scaling group (ASG). We can then easily increase or decrease the number of hosts running our application by simply changing the value of desired instances for our ASG. In addition, we will configure our ASG to place our instances evenly across our two availability zones. This ensures that if one availability zone has an outage, our application would still have half of its capacity online.

The new ASG will also:

- Automatically replace an instance if the instance fails a health check.
- Attempt to respond to availability zone outages by temporarily adding additional instances to the remaining healthy zone.
- Automatically inform CodeDeploy about new instances so that we won't have to trigger a deployment manually anymore when a new EC2 instance comes online.

Multi-phase deployments

In a production system, we have to assume that we have users continually sending requests to the system all the time. As such, when we make infrastructure or software changes, it is important to do so in such a way that causes no disruption. We must consider the effect of every change and stage the changes so that the users do not experience any loss of service. We also need to be able to roll back a change if we discover that it's not doing quite what we wanted, again without affecting our users.

CloudFormation does a good job of doing this staging for us when it can determine the relationship between resources. But not all resources have obvious

dependencies, and not all resources are strictly CloudFormation resources.

For example, in this section we'll be adding an ASG to manage our hosts and also remove the `Instance` and `Instance2` definitions that we had created for our two instances. If we were to make both of these changes simultaneously, we would likely have a gap in service between when the two hosts are terminated and when the new ASG hosts come online. Instead, we'll first add the new capacity via the ASGs, and then we will remove the old capacity only after we have checked that the new hosts are online.

The Amazon Builder's Library has a great article on making changes in deliberate phases: Ensuring rollback safety during deployments.

Adding our ASG

Adding an ASG to our CloudFormation template requires a relatively small amount of configuration. We need to reference our launch template, load balancer, and networking resources.

main.yml

```
ScalingGroup:
  Type: AWS::AutoScaling::AutoScalingGroup
  UpdatePolicy: ❶
    AutoScalingRollingUpdate:
      MinInstancesInService: "1"
      MaxBatchSize: "1"
      PauseTime: "PT15M"
      WaitOnResourceSignals: "true" ❶
      SuspendProcesses:
        - HealthCheck
        - ReplaceUnhealthy
        - AZRebalance
        - AlarmNotification
        - ScheduledActions
  Properties:
    AutoScalingGroupName: !Sub 'ASG_${AWS::StackName}'
    AvailabilityZones:
      - !Select [ 0, !GetAZs '' ]
      - !Select [ 1, !GetAZs '' ]
    MinSize: 2 ❷
    MaxSize: 6
    HealthCheckGracePeriod: 0
    HealthCheckType: ELB ❸
    LaunchTemplate: ❹
      LaunchTemplateId: !Ref InstanceLaunchTemplate
      Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber
    TargetGroupARNs:
      - !Ref LoadBalancerTargetGroup ❺
    MetricsCollection:
      -
        Granularity: "1Minute"
        Metrics:
          - "GroupMaxSize"
          - "GroupInServiceInstances"
    VPCZoneIdentifier: ❻
      - !Ref SubnetAZ1
      - !Ref SubnetAZ2
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
        PropagateAtLaunch: "true" ❼
```

❶ The `WaitOnResourceSignal` works for ASGs in the same way that the `CreationPolicy` worked on individual instances. Our launch script will get the ASG's logical ID when querying its tag, and will pass that to the `cfn-signal` command, which in turn will signal to the ASG that the instance has launched successfully.

❷ We have two availability zones, so we'll set the minimum number of hosts to two to get one instance in each.

❸ Our ASG will use our load balancer's health check to assess the health of its instances.

❹ All instances that the ASG launches will be created as per our launch template.

❺ The ASG will add all launched instances to the load balancer's target group.

❻ The VPC and subnets into which the ASG will launch the instances.

❼ Specifying `PropagateAtLaunch` ensures that this tag will be copied to all instances that are launched as part of this ASG.



If we set our desired capacity via CloudFormation, then we should never change it manually in the AWS console. If we do change it manually, CloudFormation might fail the next deployment if it finds that the current desired capacity doesn't match what it was last set to.

If you expect that you may need to make changes manually, then leave the desired capacity unset in CloudFormation.

Next, we will modify our deployment group to reference the new ASG. This will make the ASG tell CodeDeploy to deploy our application to every new instance that gets added to the ASG.

main.yml

```
  StagingDeploymentGroup:
    Type: AWS::CodeDeploy::DeploymentGroup
    Properties:
      DeploymentGroupName: staging
      AutoScalingGroups:
        - !Ref ScalingGroup
      ApplicationName: !Ref DeploymentApplication
      DeploymentConfigName: CodeDeployDefault.AllAtOnce
      ServiceRoleArn: !GetAtt DeploymentRole.Arn
      Ec2TagFilters: (1)
        - Key: aws:cloudformation:stack-name
          Type: KEY_AND_VALUE
          Value: !Ref AWS::StackName
```

(1) When the deployment group uses an ASG, we won't need `Ec2TagFilters` anymore. We will remove this property when we decommission our explicit `Instance` and `Instance2`.

Now it's time to run our `deploy-infra.sh` script to deploy our new ASG.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://awsbo-LoadB-13F2DS4LKSCVO-10652175.us-east-
1.elb.amazonaws.com:80"
]
```

Then, if we hit the load balancer endpoint, we should see our requests spread across our two explicit instances, as well as across two new instances that our ASG has spun up.

terminal

```
$  for run in {1..20}; do curl -s http://awsbo-LoadB-13F2DS4LKSCVO-
10652175.us-east-1.elb.amazonaws.com; done | sort | uniq -c
   4 Hello World from ip-10-0-113-245.ec2.internal
   5 Hello World from ip-10-0-50-202.ec2.internal
   6 Hello World from ip-10-0-61-251.ec2.internal
   5 Hello World from ip-10-0-68-58.ec2.internal
```

At this point, let's push our ASG changes to GitHub.

terminal

```
$ git add main.yml
$ git commit -m "Add ASG"
$ git push
```

Removing the explicit instances

Now that our new ASG instances are up and are serving requests through our load balancer, we can safely remove our explicit instances without causing any disruption. To do so, we just need to remove every reference to them from our `main.yml` script, and redeploy:

- The entire `Instance` resource.
- The entire `Instance2` resource.
- The entire `Targets` property from the

`LoadBalancerTargetGroup` resource.

- The `Ec2TagFilters` property from the `StagingDeploymentGroup` resource.
- The `InstanceEndpoint` and `InstanceEndpoint2` outputs.

Now, let's redeploy our infrastructure by running `deploy-infra.sh`.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://awsbo-LoadB-13F2DS4LKSCVO-10652175.us-east-
1.elb.amazonaws.com:80"
]
```

If we hit the load balancer endpoint now, we should see our traffic split between the two instances in our ASG. The other two instances have been terminated.

terminal

```
$  for run in {1..20}; do curl -s http://awsbo-LoadB-13F2DS4LKSCVO-
10652175.us-east-1.elb.amazonaws.com; done | sort | uniq -c
   9 Hello World from ip-10-0-50-202.ec2.internal
  11 Hello World from ip-10-0-68-58.ec2.internal
```

It's time to checkpoint all our changes and push them to GitHub.

terminal

```
$ git add main.yml
$ git commit -m "Remove explicitly created EC2 instances"
$ git push
```

Adding capacity in a pinch is much easier now. From the Auto Scaling console (Figure 10), you can select your ASG (Figure 11) and edit the desired capacity setting (Figure 12). The number of EC2 instances will automatically reflect your desired capacity within a few seconds.

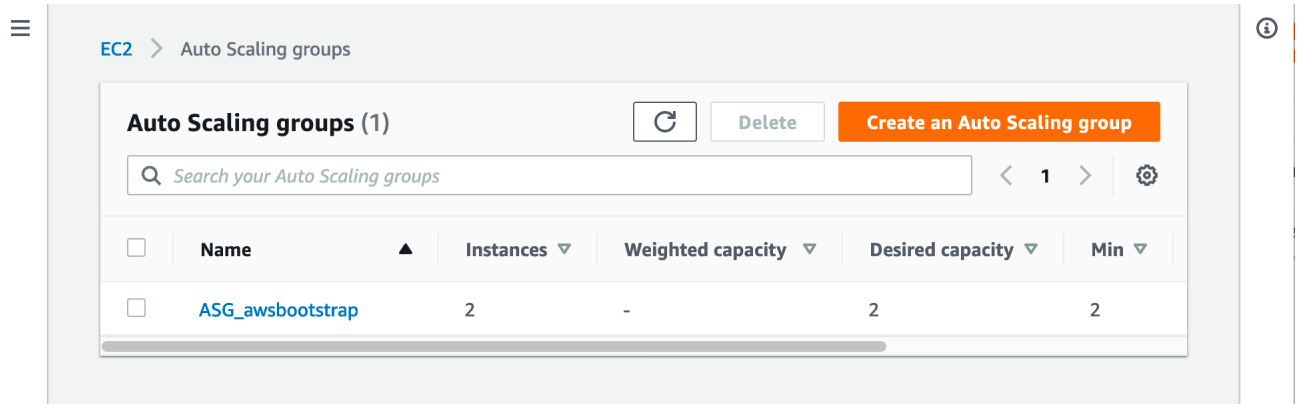


Figure 10. Auto Scaling Groups

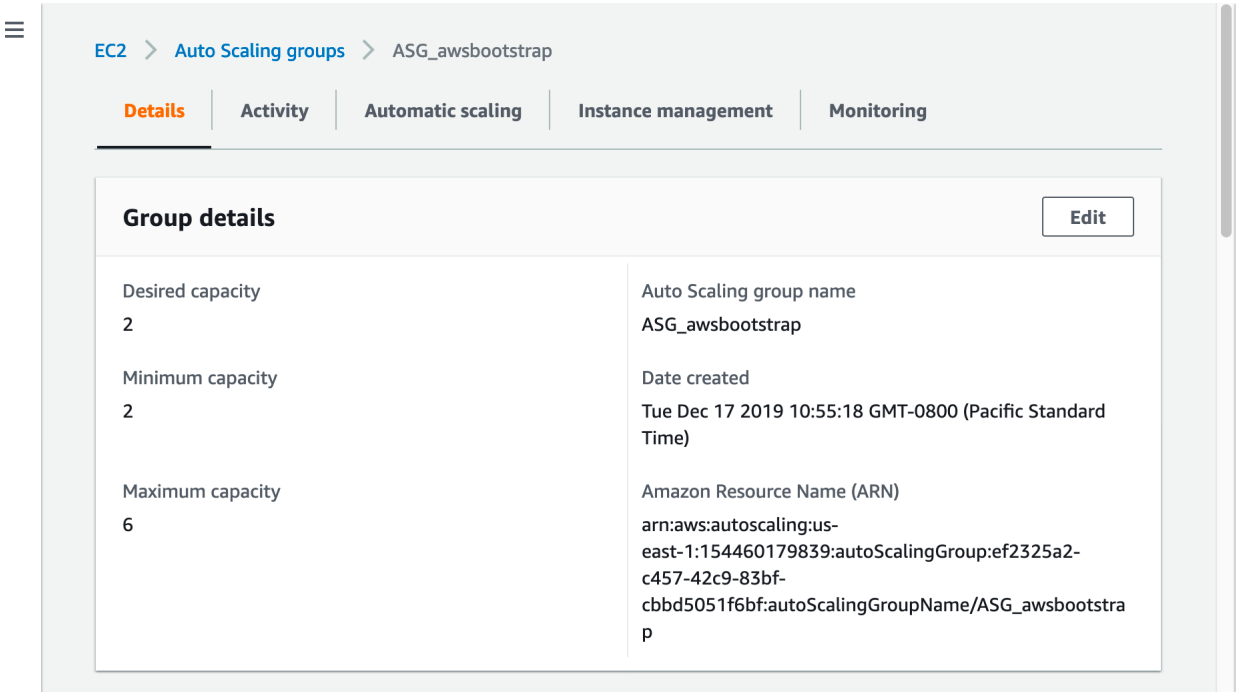


Figure 11. Auto Scaling Details

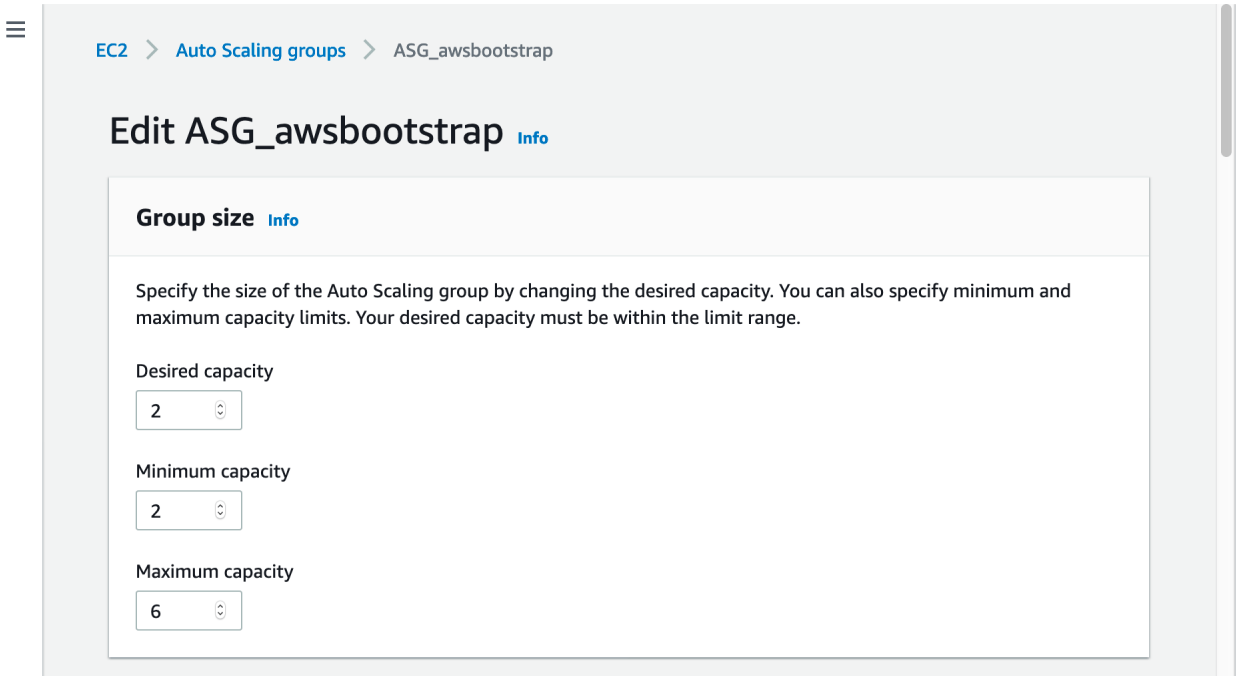


Figure 12. Auto Scaling Edit Group Size

Production

Objective

Create separate environments for staging and production.

Steps

1. Extract common resources out of `main.yml`.
2. Create separate stacks for staging and production.

In the real world, we will generally want to have at least one environment to test our application and infrastructure before rolling changes out to production. A common model is to call the testing environment 'staging' and production 'prod'. We'll set these up next.

In this section, we will decommission our existing infrastructure and replace it with two CloudFormation nested stacks representing our staging and prod environments. We will also update our CodePipeline so that it will promote updates to prod only after they've successfully passed through staging.

Adding the stack name to Hello World

Let's start by making a small change to the `start-service.sh` script so that our application will know what environment it is running in.

start-service.sh

```
#!/bin/bash -xe
source /home/ec2-user/.bash_profile
cd /home/ec2-user/app/release

# Query the EC2 metadata service for this instance's region
REGION=`curl -s http://169.254.169.254/latest/dynamic/instance-
identity/document | jq .region -r`
# Query the EC2 metadata service for this instance's instance-id
export INSTANCE_ID=`curl -s http://169.254.169.254/latest/meta-data/instance-
id`
# Query EC2 describeTags method and pull our the CFN Logical ID for this
instance
export STACK_NAME=`aws --region $REGION ec2 describe-tags \
  --filters "Name=resource-id,Values=${INSTANCE_ID}" \
            "Name=key,Values=aws:cloudformation:stack-name" \
  | jq -r ".Tags[0].Value"`

npm run start
```

Then, we also need to update the `start` script in `package.json` to pass the stack name environment variable to the application.

package.json

```
node ./node_modules/pm2/bin/pm2 start ./server.js --name hello_aws --log
../logs/app.log --update-env -- --STACK_NAME ${STACK_NAME}
```

Now, let's change the response in `server.js` to include the stack name.

server.js

```
const { hostname } = require('os');
const http = require('http');
const STACK_NAME = process.env.STACK_NAME || "Unknown Stack";
const message = `Hello World from ${hostname()} in ${STACK_NAME}\n`;
const port = 8080;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(message);
});
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname()}:${port}/`);
});
```

Let's push these changes to GitHub.

terminal

```
$ git add start-service.sh package.json server.js
$ git commit -m "Add stack name to server output"
$ git push
```

Finally, let's wait for the changes to go through the pipeline, and we should see our stack name when we hit our application's endpoint.

terminal

```
$  for run in {1..20}; do curl -s http://awsbo-LoadB-13F2DS4LKSCVO-
10652175.us-east-1.elb.amazonaws.com; done | sort | uniq -c
   9 Hello World from ip-10-0-50-202.ec2.internal in awsbootstrap
  11 Hello World from ip-10-0-68-58.ec2.internal in awsbootstrap
```

Creating a nested stack for staging

Next, we're going to move all the resources we've created for our staging environment into a separate staging stack. We'll do this by extracting the staging resources into a file

called `stage.yml`. (Here, 'stage' refers to a deployment step, not to the staging environment itself.)

To perform this split, it's easier if we start by copying the whole `main.yml` file.

```
cp main.yml stage.yml
```

Now, let's delete the following resources from `stage.yml`:

- `DeploymentRole`
- `BuildProject`
- `DeploymentApplication`
- `StagingDeploymentGroup`
- `Pipeline`
- `PipelineWebhook`

And let's delete everything that is not in the above list from `main.yml`.

We also need to delete the following input parameters from `stage.yml`:

- `CodePipelineBucket`

- `GitHubOwner`
- `GitHubRepo`
- `GitHubBranch`
- `GitHubPersonalAccessToken`

Next, we're going to add a nested stack named `Staging` to `main.yml` as an instance of our new `stage.yml` template.

main.yml

```
Staging:
  Type: AWS::CloudFormation::Stack
  Properties:
    TemplateURL: stage.yml
    TimeoutInMinutes: 30
    Parameters:
      EC2InstanceType: !Ref EC2InstanceType
      EC2AMI: !Ref EC2AMI
```

Now we need to add outputs in `stage.yml` so that the `main.yml` stack knows about the load balancer endpoints and the ASG of the stage.

stage.yml

```
Outputs:
  LBEndpoint:
    Description: The DNS name for the LB
    Value: !Sub "http://${LoadBalancer.DNSName}:80"
  ScalingGroup:
    Description: The ScalingGroup for this stage
    Value: !Ref ScalingGroup
```



We don't need `Export` properties in `stage.yml`. This is because `stage.yml` will be referenced only by the `main.yml` stack, and parent stacks can access the output variables of nested stacks directly.

Now we need to change the `StagingDeploymentGroup` resource to refer to the output from the staging nested stack.

main.yml

```
StagingDeploymentGroup:
  Type: AWS::CodeDeploy::DeploymentGroup
  Properties:
    DeploymentGroupName: staging
    AutoScalingGroups:
    - !GetAtt Staging.Outputs.ScalingGroup ❶
    ApplicationName: !Ref DeploymentApplication
    DeploymentConfigName: CodeDeployDefault.AllAtOnce
    ServiceRoleArn: !GetAtt DeploymentRole.Arn
```

❶ Refers to the ASG that the staging stack returns.

We also need to change the endpoint that `main.yml` returns.

main.yml

```
StagingLBEndpoint:
  Description: The DNS name for the staging LB
  Value: !GetAtt Staging.Outputs.LBEndpoint
  Export:
    Name: StagingLBEndpoint
```

At this point we need to deal with one of CloudFormation's quirks. Nested stacks must be referenced as S3 URLs. To deal with this, we can use

CloudFormation packaging to help us upload and transform our templates.

But first we'll need an S3 bucket to store our packaged templates. This is another thing that can go into our `setup.yml` template, so let's add the input parameter first.

setup.yml

```
CloudFormationBucket:
  Type: String
  Description: 'The S3 bucket for CloudFormation templates.'
```

And then let's add the resource for the S3 bucket.

setup.yml

```
CloudFormationS3Bucket:
  Type: AWS::S3::Bucket
  DeletionPolicy: Retain
  Properties:
    BucketName: !Ref CloudFormationBucket
    PublicAccessBlockConfiguration:
      BlockPublicAcls: true
      BlockPublicPolicy: true
      IgnorePublicAcls: true
      RestrictPublicBuckets: true
    BucketEncryption:
      ServerSideEncryptionConfiguration:
        - ServerSideEncryptionByDefault:
            SSEAlgorithm: AES256
```

Next, we'll add an environment variable in `deploy-infra.sh` to define the S3 bucket name for the packaged CloudFormation templates.

deploy-infra.sh

```
CFN_BUCKET="$STACK_NAME-cfn-$AWS_ACCOUNT_ID"
```

And finally, we're going to pass the bucket name as a parameter when we deploy `setup.yml`.

deploy-infra.sh

```
# Deploys static resources
echo -e "\n\n============ Deploying setup.yml ============"
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME-setup \
  --template-file setup.yml \
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    CodePipelineBucket=$CODEPIPELINE_BUCKET \
    CloudFormationBucket=$CFN_BUCKET ❶
```

❶ Pass in the bucket used to store packaged CloudFormation resources.

Between the deploy commands for `setup.yml` and `main.yml` in our `deploy-infra.sh` script, we also need to add a new set of commands to package our nested stacks.

deploy-infra.sh

```
# Package up CloudFormation templates into an S3 bucket
echo -e "\n\n=========== Packaging main.yml ==========="
mkdir -p ./cfn_output

PACKAGE_ERR="$(aws cloudformation package \
  --region $REGION \
  --profile $CLI_PROFILE \
  --template main.yml \
  --s3-bucket $CFN_BUCKET \
  --output-template-file ./cfn_output/main.yml 2>&1)" ❶

if ! [[ $PACKAGE_ERR =~ "Successfully packaged artifacts" ]]; then
  echo "ERROR while running 'aws cloudformation package' command:"
  echo $PACKAGE_ERR
  exit 1
fi
```

❶ This will write the packaged CloudFormation template to `/cfn_output/main.yml`.

We now need to change the deploy command for `main.yml` in `deploy-infra.sh` to refer to the packaged template file.

deploy-infra.sh

```
# Deploy the CloudFormation template
echo -e "\n\n=========== Deploying main.yml ==========="
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME \
  --template-file ./cfn_output/main.yml \ ❶
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    EC2InstanceType=$EC2_INSTANCE_TYPE \
    GitHubOwner=$GH_OWNER \
    GitHubRepo=$GH_REPO \
    GitHubBranch=$GH_BRANCH \
    GitHubPersonalAccessToken=$GH_ACCESS_TOKEN \
    CodePipelineBucket=$CODEPIPELINE_BUCKET
```

❶ The output of the `aws cloudformation package` command.

Finally, we need to change the section of `deploy-infra.sh` that prints the endpoint URLs so that it catches both our staging endpoint, as well as the forthcoming prod endpoint.

deploy-infra.sh

```
# If the deploy succeeded, show the DNS name of the endpoints
if [ $? -eq 0 ]; then
  aws cloudformation list-exports \
    --profile awsbootstrap \
    --query "Exports[?ends_with(Name,'LBEndpoint')].Value"
fi
```

Now it's time to deploy our changes.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Packaging main.yml ===========


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://awsbo-LoadB-1SN04P0UGU5RV-1429520787.us-east-
1.elb.amazonaws.com:80"
]
```

Within a few minutes we should have all the resources recreated as they were, but organized under our new

staging stack.

At this point, if we go to the CloudFormation console we should see three stacks.

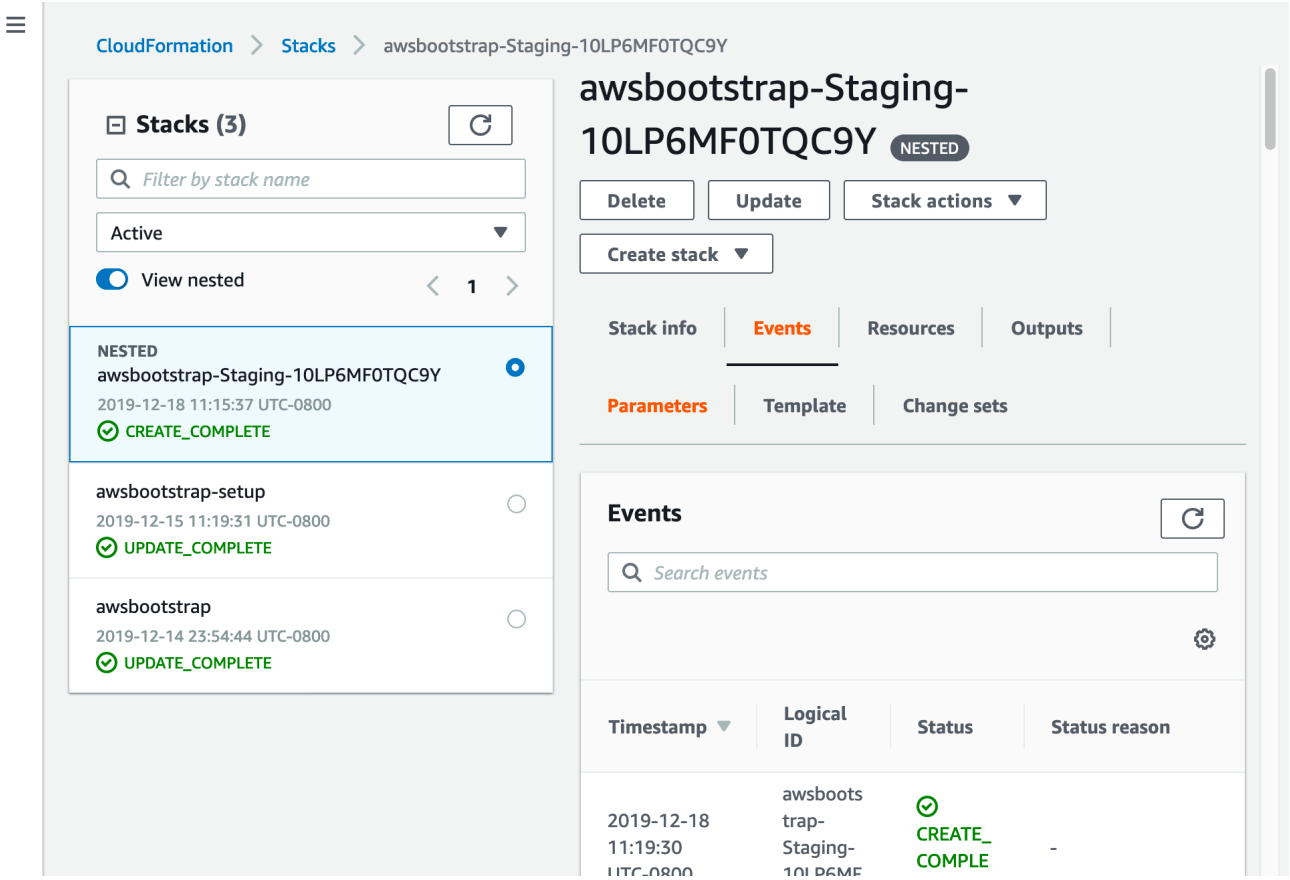


Figure 13. New Staging nested stack

`awsbootstrap-setup`

A root stack containing our S3 buckets for CodePipeline and CloudFormation.

`awsbootstrap`

A root stack for our application containing our deployment resources and our staging nested stack.

awsbootstrap-Staging-XYZ

Our new nested staging stack containing all the application resources.

Let's verify that everything is still working.

terminal

```
$ for run in {1..20}; do curl -s http://awsbo-LoadB-1SN04P0UGU5RV-
1429520787.us-east-1.elb.amazonaws.com; done | sort | uniq -c
  10 Hello World from ip-10-0-102-103.ec2.internal in awsbootstrap-Staging-
10LP6MF0TQC9Y
  10 Hello World from ip-10-0-61-182.ec2.internal in awsbootstrap-Staging-
10LP6MF0TQC9Y
```

And now it's time to commit our changes to GitHub.

terminal

```
$ git add main.yml stage.yml setup.yml deploy-infra.sh
$ git commit -m "Split out staging nested stack"
$ git push
```

Adding the prod stage

Now it's time to add our prod stack. Extracting the staging stack out of the main template was most of the work, so adding prod is going to be quite straightforward.

First, let's add a deployment group so that our production hosts can be configured for deployment.

main.yml

```
ProdDeploymentGroup:
  Type: AWS::CodeDeploy::DeploymentGroup
  Properties:
    DeploymentGroupName: prod
    AutoScalingGroups:
      - !GetAtt Prod.Outputs.ScalingGroup
    ApplicationName: !Ref DeploymentApplication
    DeploymentConfigName: CodeDeployDefault.OneAtATime ❶
    ServiceRoleArn: !GetAtt DeploymentRole.Arn
```

❶ We could have chosen to put the deployment groups in `stage.yml`, but it is often useful to have different deployment configurations for different stages. For example, here we set the prod deployment to go on one host at a time, whereas the staging deployment was set to go all at once.

Next, we'll add a new stage in our `Pipeline` resource called `Prod` at the end of the list.

main.yml

```
- Name: Prod
  Actions:
    - Name: Prod
      InputArtifacts:
        - Name: Build
      ActionTypeId:
        Category: Deploy
        Owner: AWS
        Version: 1
        Provider: CodeDeploy
      Configuration:
        ApplicationName: !Ref DeploymentApplication
        DeploymentGroupName: !Ref ProdDeploymentGroup
      RunOrder: 1
```

Then we'll add the new nested stack to `main.yml`.

main.yml

```
Prod:
  Type: AWS::CloudFormation::Stack
  DependsOn: Staging ❶
  Properties:
    TemplateURL: stage.yml
    TimeoutInMinutes: 30
    Parameters:
      EC2InstanceType: !Ref EC2InstanceType
      EC2AMI: !Ref EC2AMI
```

❶ Updates to the prod stack will not be enacted until the staging stack successfully applies stack updates.

Finally, we need to add an output in `main.yml` to return the prod endpoint.

main.yml

```
ProdLBEndpoint:
  Description: The DNS name for the prod LB
  Value: !GetAtt Prod.Outputs.LBEndpoint
  Export:
    Name: ProdLBEndpoint
```

And now let's deploy.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Packaging main.yml ===========


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://awsbo-LoadB-1DDGVDL6XEI9S-615344013.us-east-
1.elb.amazonaws.com:80",
    "http://awsbo-LoadB-1SN04P0UGU5RV-1429520787.us-east-
1.elb.amazonaws.com:80"
]
```

The staging endpoint should still be working when the deployment finishes, but our new production endpoint won't be yet. This is because the deployment groups are configured so that the ASG will launch all new instances with the most recent revision deployed. But there will be no recent revision until the first proper deployment occurs. To fix this, we need to go to our pipeline and click *Release Changes* in order to get a deployment to prod. Once deployment finishes, then the prod hosts should start responding as well.

terminal

```
$ for run in {1..20}; do curl -s http://awsbo-LoadB-1DDGVDL6XEI9S-
615344013.us-east-1.elb.amazonaws.com; done | sort | uniq -c
  10 Hello World from ip-10-0-46-233.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
  10 Hello World from ip-10-0-77-238.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
```

Let's wrap this up by pushing these changes to our GitHub repository.

terminal

```
$ git add main.yml
$ git commit -m "Add prod nested stack"
$ git push
```

Custom Domains

Objective

Access our application from a custom domain.

Steps

1. Register a domain with Route 53.
2. Create a DNS hosted zone for our domain.
3. Map our domain to the load balancers.

In this section, we will walk through the process of registering a domain with Route 53, and making our application use it. If you already own a domain that you want to use, you can migrate your DNS to Route 53 or completely transfer control of your domain and DNS to Route 53.



Domain name registration costs on the order of tens of dollars yearly, and Route 53 hosted zones are $0.50 per month. Promotional credits are not applicable to either expense. If you don't want to incur these fees just for experimentation, you can skip this section and the next one (HTTPS) and continue using the AWS-provided DNS names of your load balancers.

Registering a domain

Registering a domain is done infrequently and requires some human intervention. Therefore, we will do this manually through the Route 53 console. After we've chosen our domain name, Route 53 will check if it is available, and if it is, we can proceed with the registration.

Once we have completed the registration process, Route 53 will put our domain under *Pending Requests*. After a few minutes, it should show up in the *Registered Domains* view.

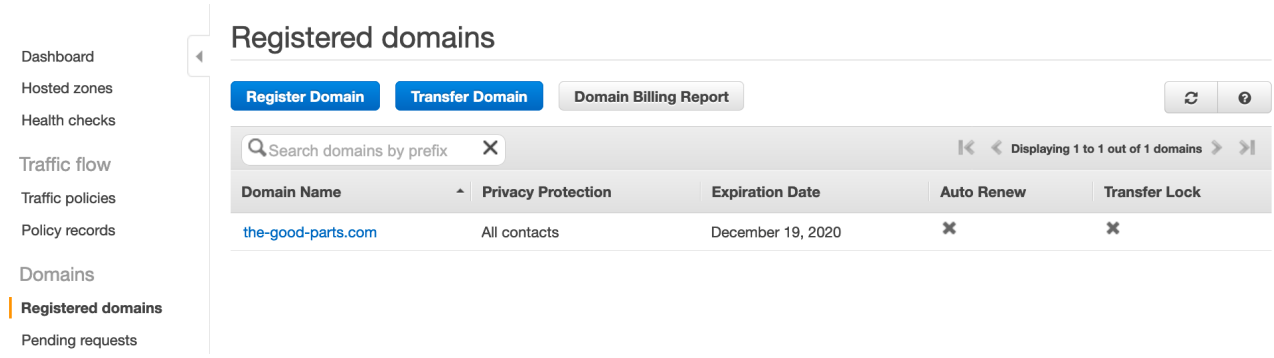


Figure 14. Registered Domains

Getting our hosted zone

Once our domain is registered, Route 53 will automatically create a DNS hosted zone for us.

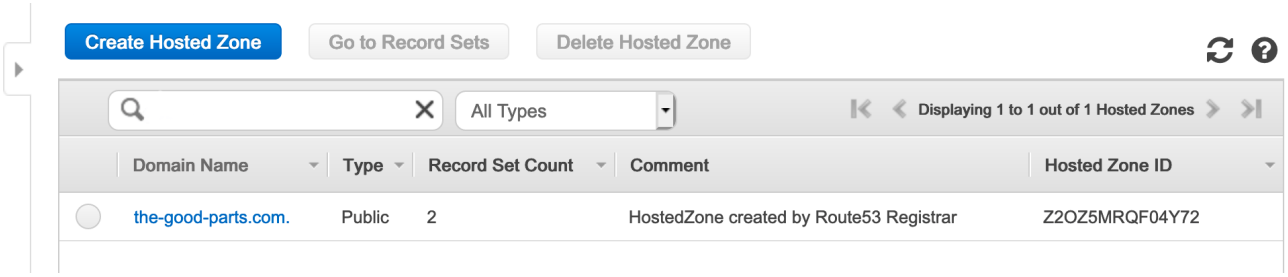


Figure 15. Hosted Zones

When we click on the hosted zone, we should see the default NS and SOA records. We will use CloudFormation to add other records to our hosted zone.

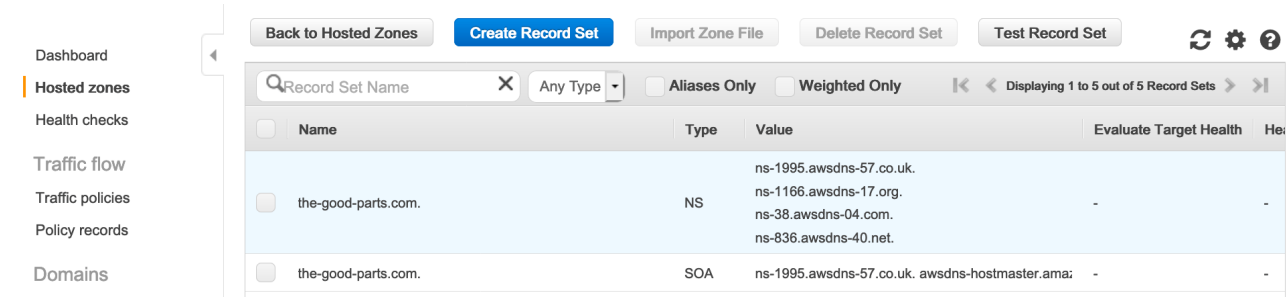


Figure 16. Hosted Zone

Creating a DNS record per stage

Let's start by adding two new input parameters in our `stage.yml` to receive the stage domain and subdomain.

stage.yml

```
Domain:
  Type: String
SubDomain:
  Type: String
```

Then, let's add a resource to create a Route 53 A record that points `<subdomain>.<domain>` to the load balancer.

stage.yml

```
DNS:
  Type: AWS::Route53::RecordSet
  Properties:
    HostedZoneName: !Sub '${Domain}.'
    Name: !Sub '${SubDomain}.${Domain}.'
    Type: A
    AliasTarget:
      HostedZoneId: !GetAtt LoadBalancer.CanonicalHostedZoneID
      DNSName: !GetAtt LoadBalancer.DNSName
```

Next, let's change the stage output to return the URL with our custom domain rather than the load balancer's default endpoint.

stage.yml

```
LBEndpoint:
  Description: The DNS name for the stage
  Value: !Sub "http://${DNS}"
```

Then we also need to add an input parameter in `main.yml` to receive our custom domain name.

main.yml

```
Domain:
  Type: String
```

And finally, we need to pass our custom domain to the nested stacks.

main.yml

```
Staging:
  Type: AWS::CloudFormation::Stack
  Properties:
    TemplateURL: stage.yml
    TimeoutInMinutes: 30
    Parameters:
      EC2InstanceType: !Ref EC2InstanceType
      EC2AMI: !Ref EC2AMI
      Domain: !Ref Domain ❶
      SubDomain: staging ❷

Prod:
  Type: AWS::CloudFormation::Stack
  Properties:
    TemplateURL: stage.yml
    TimeoutInMinutes: 30
    Parameters:
      EC2InstanceType: !Ref EC2InstanceType
      EC2AMI: !Ref EC2AMI
      Domain: !Ref Domain ❶
      SubDomain: prod ❷
```

❶ Passes the domain name to the nested stack.

❷ Passes a stack-specific subdomain to the nested stack.

At this point, let's add our domain name as an environment variable at the top of `deploy-infra.sh`.

deploy-infra.sh

```
DOMAIN=the-good-parts.com ❶
```

❶ Replace with your domain name.

And now we can pass our domain to the CloudFormation template.

deploy-infra.sh

```
# Deploy the CloudFormation template
echo -e "\n\n=========== Deploying main.yml ==========="
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME \
  --template-file ./cfn_output/main.yml \
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    EC2InstanceType=$EC2_INSTANCE_TYPE \
    Domain=$DOMAIN \ ❶
    GitHubOwner=$GH_OWNER \
    GitHubRepo=$GH_REPO \
    GitHubBranch=$GH_BRANCH \
    GitHubPersonalAccessToken=$GH_ACCESS_TOKEN \
    CodePipelineBucket=$CODEPIPELINE_BUCKET
```

❶ Passes the domain name to `main.yml`.

Let's deploy to see our custom domain in action.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Packaging main.yml ===========


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://prod.the-good-parts.com",
    "http://staging.the-good-parts.com"
]
```

And now have a much more human-friendly endpoint for our two stages. We should also be able to see the A records in our Route 53 hosted zone.

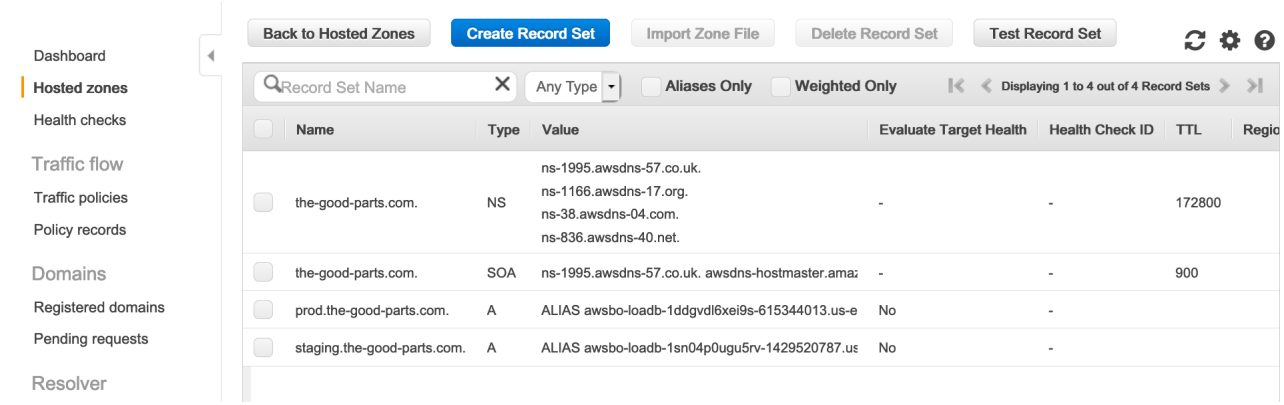


Figure 17. New A Records

The DNS propagation can take a few minutes. After a while, we should be able to reach our application through our custom domain.

terminal

```
$ for run in {1..20}; do curl -s staging.the-good-parts.com; done | sort |
uniq -c
   9 Hello World from ip-10-0-102-103.ec2.internal in awsbootstrap-Staging-
10LP6MF0TQC9Y
  11 Hello World from ip-10-0-61-182.ec2.internal in awsbootstrap-Staging-
10LP6MF0TQC9Y
```

terminal

```
$ for run in {1..20}; do curl -s prod.the-good-parts.com; done | sort | uniq
-c
  10 Hello World from ip-10-0-46-233.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
  10 Hello World from ip-10-0-77-238.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
```



If the `curl` commands work, but your browser times out trying to connect, it may be trying to upgrade to HTTPS in order to provide better security. You can try from another browser or wait until we enable HTTPS in the next section.

Now we can commit all our changes to checkpoint our progress.

terminal

```
$ git add deploy-infra.sh main.yml stage.yml
$ git commit -m "Add a custom domain"
$ git push
```

HTTPS

Objective

Migrate our endpoint from HTTP to HTTPS.

Steps

1. Manually create a TLS certificate.
2. Add an HTTPS endpoint.
3. Make the application speak HTTPS.
4. Remove the HTTP endpoint.

As things stand, our application is responding to unencrypted HTTP traffic. In the real world, we want to protect any data as it traverses the network. To do that, we must encrypt our traffic and serve it over HTTPS.

We'll also take this as an opportunity to practice the two-phase change process discussed in Multi-phase deployments to give the chance to anyone using our HTTP endpoint to migrate to HTTPS before we turn off HTTP.

Creating the certificate

Requesting a certificate is an infrequent operation that requires human intervention for validation (or more automation than makes sense, for a process that happens only once). Therefore, we're going to create our certificate manually. To start, let's visit the AWS Certificate Manager (ACM) console and hit *Request a certificate*. Then, let's select the public certificate option.

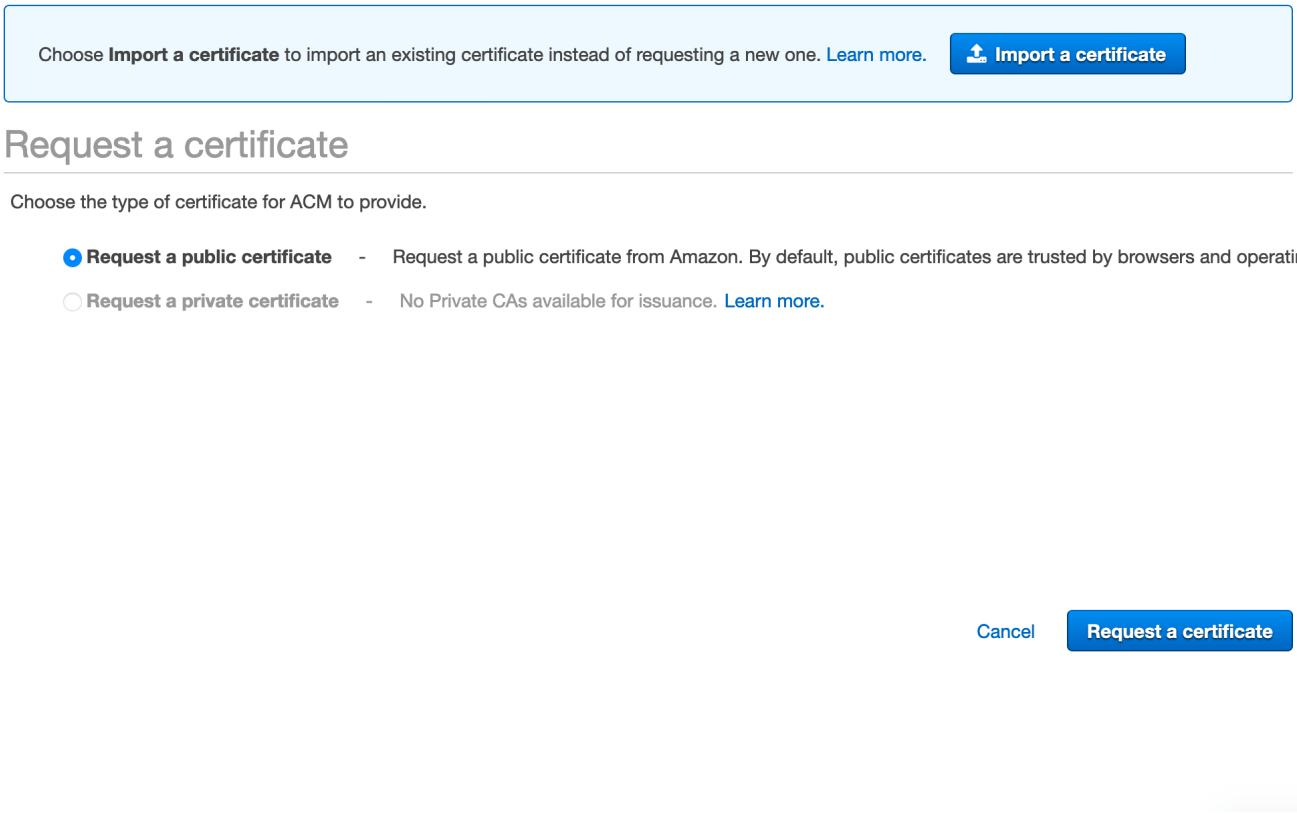


Figure 18. Request a Certificate

Next, let's enter our bare domain (e.g., `the-good-parts.com`) as well as a wildcard version of the domain (e.g., `*.the-good-parts.com`). The wildcard will cover our prod and staging subdomains.

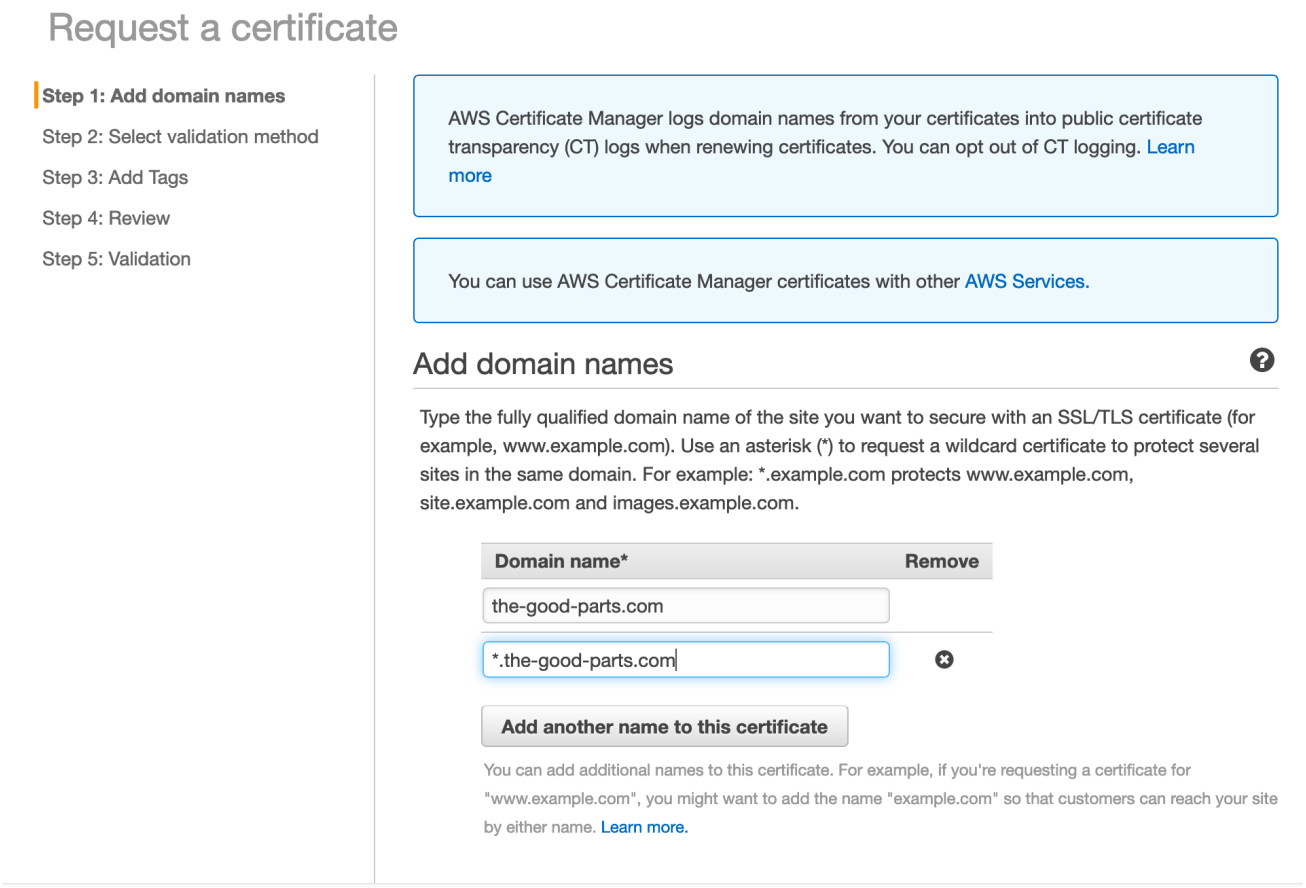


Figure 19. Add Domain Names

Now, we must validate that we control the domain.

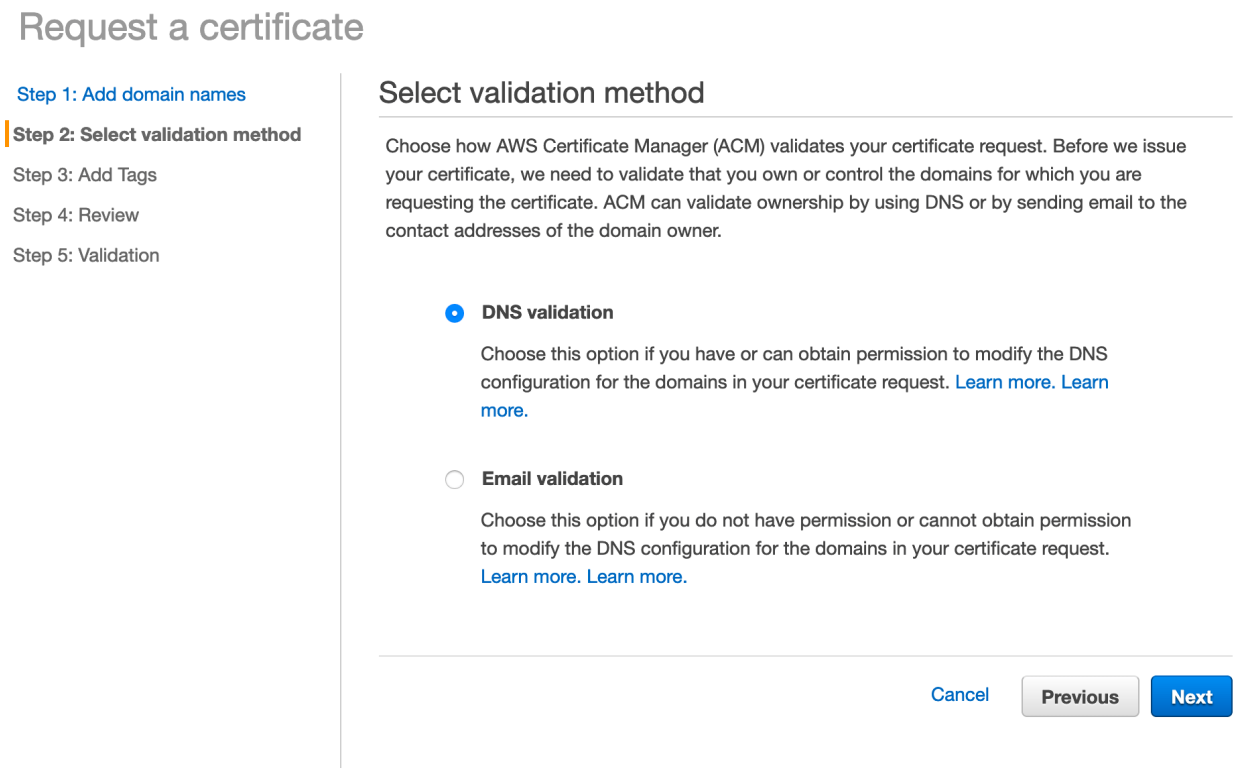


Figure 20. Select Validation Method

If you chose DNS *validation*, you will reach a *Validation* step that asks you to add a CNAME record to your DNS hosted zone. If you registered your domain through Route 53, you can simply click the *Create record in Route* 53 button to complete the validation process. Otherwise, you have to add the requested record to your DNS hosting service.

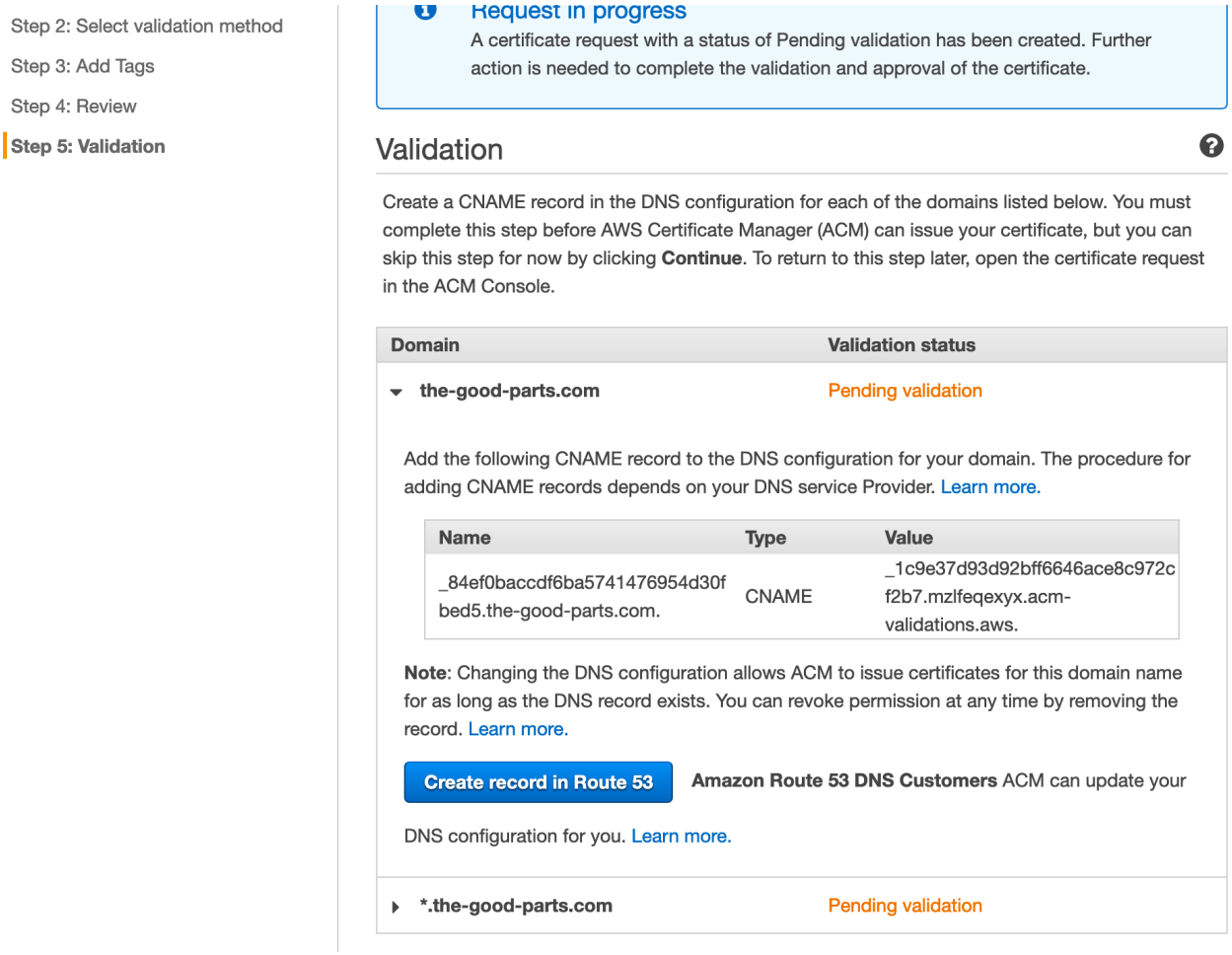


Figure 21. Create CNAME Records

It usually takes a few minutes for the certificate to be validated. Once it is validated, you should see your issued certificate in the ACM console.

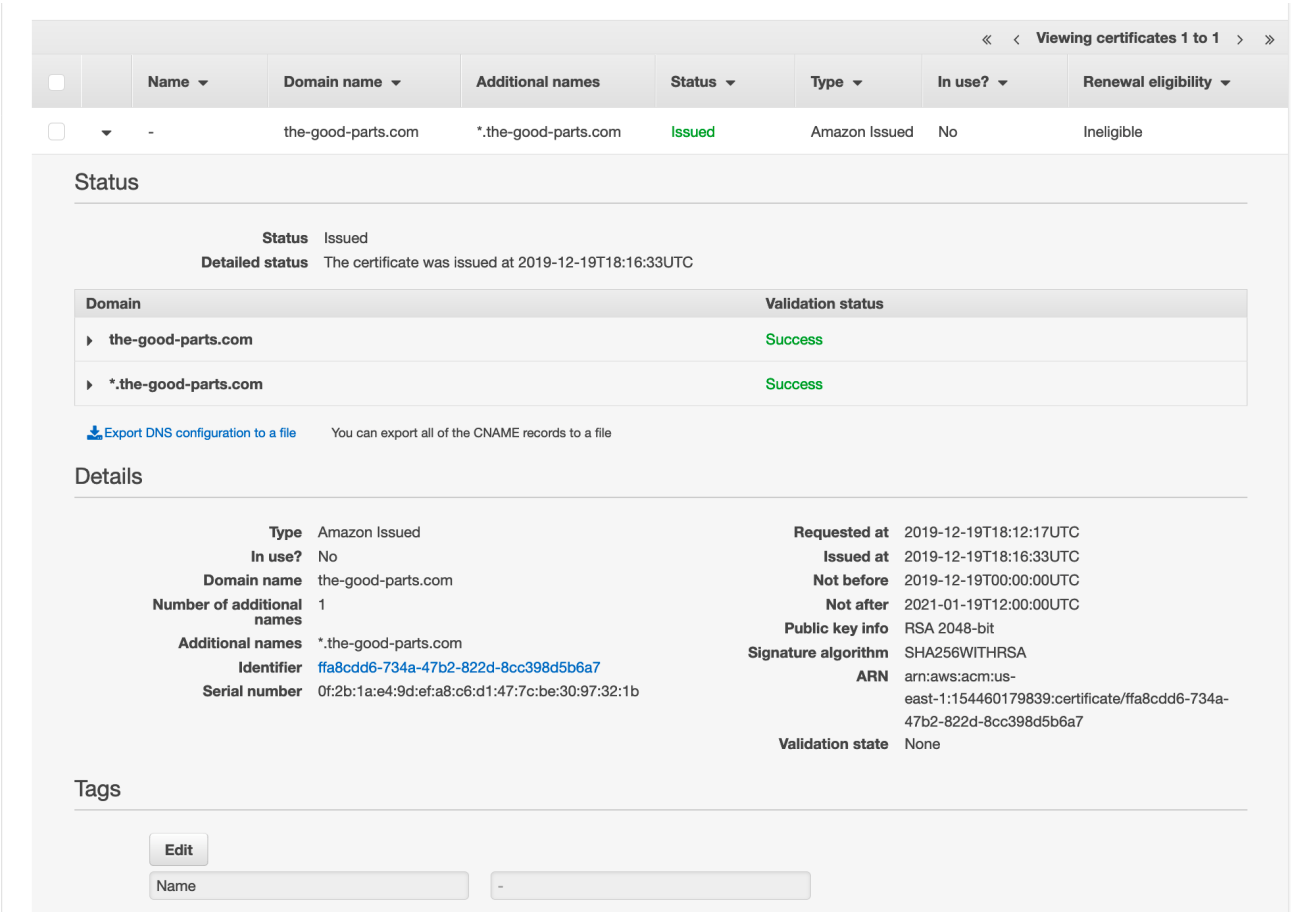


Figure 22. Validated Certificate

You can also inspect the CNAME record that was added to your hosted zone in Route 53.

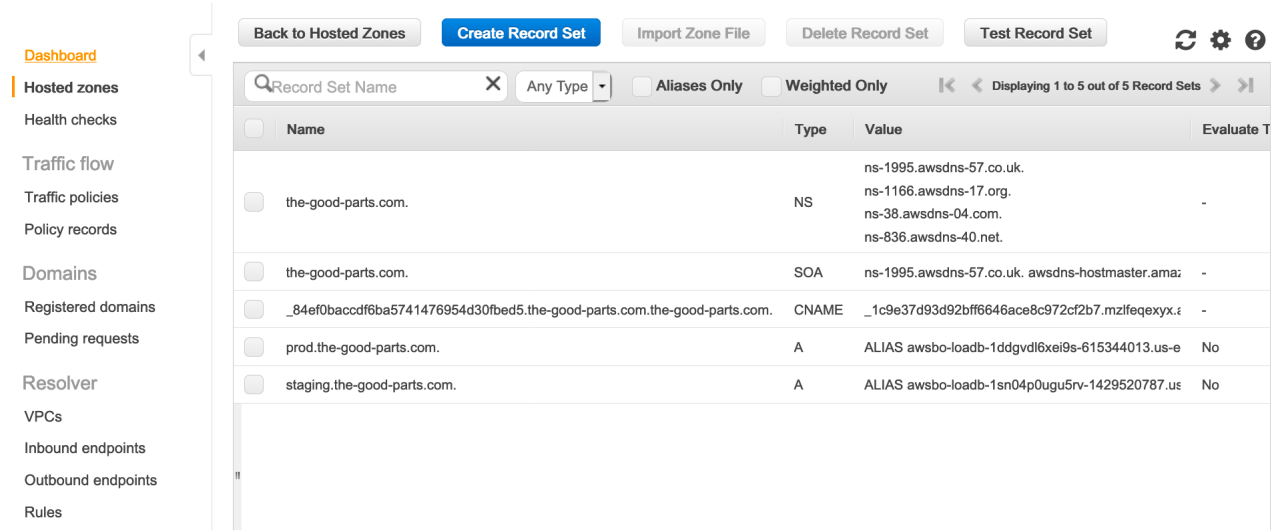


Figure 23. Hosted Zone CNAME Record

Adding the HTTPS endpoint

We will now update our `deploy-infra.sh` script to retrieve the certificate ARN. This should go at the top of the script, and depends on the `DOMAIN` environment variable.

deploy-infra.sh

```
DOMAIN=the-good-parts.com
CERT=`aws acm list-certificates --region $REGION --profile awsbootstrap
--output text \
        --query "CertificateSummaryList[?DomainName=='
$DOMAIN'].CertificateArn | [0]"` ❶
```

❶ Newly added environment variable holding our certificate.

We then have to pass the certificate ARN as a parameter to `main.yml`.

deploy-infra.sh

```
# Deploy the CloudFormation template
echo -e "\n\n============ Deploying main.yml ============"
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME \
  --template-file ./cfn_output/main.yml \
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    EC2InstanceType=$EC2_INSTANCE_TYPE \
    Domain=$DOMAIN \
    Certificate=$CERT \ ❶
    GitHubOwner=$GH_OWNER \
    GitHubRepo=$GH_REPO \
    GitHubBranch=$GH_BRANCH \
    GitHubPersonalAccessToken=$GH_ACCESS_TOKEN \
    CodePipelineBucket=$CODEPIPELINE_BUCKET
```

❶ The certificate ARN.

We also have to add this as a parameter in the `main.yml` template.

main.yml

```
Certificate:
  Type: String
  Description: 'An existing ACM certificate ARN for your domain'
```

Then, we also have to pass the ARN to our nested stacks by adding a parameter to the `Staging` and `Prod` resources in `main.yml`.

main.yml

```
Staging:
  Type: AWS::CloudFormation::Stack
  Properties:
    TemplateURL: stage.yml
    TimeoutInMinutes: 30
    Parameters:
      EC2InstanceType: !Ref EC2InstanceType
      EC2AMI: !Ref EC2AMI
      Domain: !Ref Domain
      SubDomain: staging
      Certificate: !Ref Certificate ❶

Prod:
  Type: AWS::CloudFormation::Stack
  Properties:
    TemplateURL: stage.yml
    TimeoutInMinutes: 30
    Parameters:
      EC2InstanceType: !Ref EC2InstanceType
      EC2AMI: !Ref EC2AMI
      Domain: !Ref Domain
      SubDomain: prod
      Certificate: !Ref Certificate ❶
```

❶ The certificate ARN.

Finally, we have to add an input parameter in `stage.yml`

to receive the certificate ARN from `main.yml`.

stage.yml

```
Certificate:
  Type: String
  Description: 'An existing ACM certificate ARN for subdomain.domain'
```

Next, we're going to modify our security group to allow traffic on HTTPS ports 443 and 8443.

stage.yml

```
SecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId: !Ref VPC
    GroupDescription:
      !Sub 'Internal Security group for ${AWS::StackName}'
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 8080
        ToPort: 8080
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp ❶
        FromPort: 8443
        ToPort: 8443
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 80
        ToPort: 80
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp ❶
        FromPort: 443
        ToPort: 443
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: 0.0.0.0/0
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ Newly added HTTPS ports.

At this point, we need to modify the `UserData` section of our EC2 launch template to make the instance generate a self-signed certificate automatically when it starts up. This certificate will be used for traffic between the load balancer and the instance.

stage.yml

```
cat > /tmp/install_script.sh << EOF
  # START
  echo "Setting up NodeJS Environment"
  curl https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash

  # Dot source the files to ensure that variables are available within the
current shell
  . /home/ec2-user/.nvm/nvm.sh
  . /home/ec2-user/.bashrc

  # Install NVM, NPM, Node.JS
  nvm alias default v12.7.0
  nvm install v12.7.0
  nvm use v12.7.0

  # Create log directory
  mkdir -p /home/ec2-user/app/logs

  # Create a self-signed TLS certificate to communicate with the load
balancer
  mkdir -p /home/ec2-user/app/keys
  cd /home/ec2-user/app/keys
  openssl req -new -newkey rsa:4096 -days 365 -nodes -x509 \ ❶
              -subj "/C=/ST=/L=/O=/CN=localhost" -keyout key.pem -out
cert.pem
EOF
```

❶ Generates a certificate (`cert.pem`) and private key (`key.pem`) and puts them in `/home/ec-user/app/keys`.

Next, we add a new target group so that the load balancer forwards traffic to the application's 8443 port.

stage.yml

```
HTTPSLoadBalancerTargetGroup:
  Type: AWS::ElasticLoadBalancingV2::TargetGroup
  Properties:
    TargetType: instance
    Port: 8443 ❶
    Protocol: HTTPS
    VpcId: !Ref VPC
    HealthCheckEnabled: true
    HealthCheckProtocol: HTTPS ❷
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ 8443 is the non-privileged port that our application will use to serve HTTPS requests.

❷ The health check will also be made on the HTTPS port.

Now, let's add a new load balancer listener for HTTPS.

stage.yml

```
HTTPSLoadBalancerListener:
  Type: AWS::ElasticLoadBalancingV2::Listener
  Properties:
    DefaultActions:
      - Type: forward
        TargetGroupArn: !Ref HTTPSLoadBalancerTargetGroup
    LoadBalancerArn: !Ref LoadBalancer
    Certificates:
      - CertificateArn: !Ref Certificate ❶
    Port: 443 ❷
    Protocol: HTTPS
```

❶ The certificate ARN.

❷ 443 is the standard HTTPS port.

Then we need to add the new HTTPS target group to the `ScalingGroup` ASG so that the instances managed by the ASG will be added automatically behind the load

balancer's HTTPS target.

stage.yml

```
TargetGroupARNs:
  - !Ref LoadBalancerTargetGroup
  - !Ref HTTPSLoadBalancerTargetGroup (1)
```

(1) References the new HTTPS target group.

Next, we will also add a new entry to the `Outputs` section in `stage.yml` to return the URL for our new HTTPS endpoint.

stage.yml

```
HTTPSEndpoint:
  Description: The DNS name for the stage
  Value: !Sub "https://${DNS}"
```

Finally, we'll add two new outputs from `main.yml` for the new HTTPS endpoints.

main.yml

```
Outputs:
  StagingLBEndpoint:
    Description: The DNS name for the staging LB
    Value: !GetAtt Staging.Outputs.LBEndpoint
    Export:
      Name: StagingLBEndpoint
  StagingHTTPSLBEndpoint: (1)
    Description: The DNS name for the staging HTTPS LB
    Value: !GetAtt Staging.Outputs.HTTPSEndpoint
    Export:
      Name: StagingHTTPSLBEndpoint
  ProdLBEndpoint:
    Description: The DNS name for the prod LB
    Value: !GetAtt Prod.Outputs.LBEndpoint
    Export:
      Name: ProdLBEndpoint
  ProdHTTPSLBEndpoint: (1)
    Description: The DNS name for the prod HTTPS LB
    Value: !GetAtt Prod.Outputs.HTTPSEndpoint
    Export:
      Name: ProdHTTPSLBEndpoint
```

(1) Newly added HTTPS endpoints.

It's time to deploy our changes. This change may take longer than previous updates, because it has to spin up two new instances per stage with the updated launch script, and then terminate the old ones.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Packaging main.yml ===========


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://prod.the-good-parts.com",
    "https://prod.the-good-parts.com",
    "http://staging.the-good-parts.com",
    "https://staging.the-good-parts.com"
]
```

Our HTTP endpoints should continue to respond correctly. However, if we try to reach the new HTTPS endpoints, we'll get an error, because the load balancer can't yet reach our application on port 8443.

terminal

```
curl https://prod.the-good-parts.com
<html>
<head><title>502 Bad Gateway</title></head>
<body bgcolor="white">
<center><h1>502 Bad Gateway</h1></center>
</body>
</html>
```

If you were to look for the new HTTPS target group in the AWS console, you should see no healthy hosts in the

Monitoring tab. You can also see that the EC2 instances are being continuously created and destroyed.

This is happening because we haven't yet updated our application to serve HTTPS requests on port 8443, so our instances are failing their health checks. In the real world, it would have been better to update the application first, and only then update the infrastructure. But here, we wanted to do it in the reverse order to demonstrate the behavior of the load balancer health checks. So, let's push our infrastructure changes to GitHub, and then let's fix our application.

terminal

```
$ git add deploy-infra.sh main.yml stage.yml
$ git commit -m "Add HTTPS listener; Add cert to launch script"
$ git push
```

Making our application speak HTTPS

Let's update our application to listen to HTTPS requests on port 8443. We're going to make the application use the self-signed certificate that the EC2 instance generated in the `UserData` script when it came online.

server.js

```
const { hostname } = require('os');
const http = require('http');
const https = require('https'); ❶
const fs = require('fs');

const STACK_NAME = process.env.STACK_NAME || "Unknown Stack";
const port = 8080;
const httpsPort = 8443; ❷
const httpsKey = '../keys/key.pem' ❸
const httpsCert = '../keys/cert.pem' ❸

if (fs.existsSync(httpsKey) && fs.existsSync(httpsCert)) { ❹
  console.log('Starting https server')
  const message = `Hello HTTPS World from ${hostname()} in ${STACK_NAME}\n`;
  const options = { key: fs.readFileSync(httpsKey), cert:
fs.readFileSync(httpsCert) };
  const server = https.createServer(options, (req, res) => { ❺
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end(message);
  });
  server.listen(httpsPort, hostname, () => {
    console.log(`Server running at http://${hostname()}:${httpsPort}/`);
  });
}

console.log('Starting http server')
const message = `Hello HTTP World from ${hostname()} in ${STACK_NAME}\n`;
const server = http.createServer((req, res) => { ❻
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(message);
});
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname()}:${port}/`);
});
```

❶ Import the `https` node library.

❷ We'll use 8443 as our local HTTPS port number.

❸ Path to the key and certificate generated in the instance launch script.

❹ For a graceful roll-out, we'll check for the key and certificate before launching the HTTPS server.

❺ We launch an HTTPS server if we have a key and certificate.

❻ We continue to launch our HTTP server until all traffic has been

migrated.

And let's push this change to GitHub.

terminal

```
$ git add server.js
$ git commit -m "Add HTTPS listener to the application"
$ git push
```

Now let's wait for the deployment to go through. Our HTTP and HTTPS endpoints should then start working.

terminal

```
$ for run in {1..20}; do curl -s http://staging.the-good-parts.com; done |
sort | uniq -c
  10 Hello HTTP World from ip-10-0-12-230.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
  10 Hello HTTP World from ip-10-0-85-169.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
```

terminal

```
$ for run in {1..20}; do curl -s https://staging.the-good-parts.com; done |
sort | uniq -c
  10 Hello HTTPS World from ip-10-0-12-230.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
  10 Hello HTTPS World from ip-10-0-85-169.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
```

terminal

```
$ for run in {1..20}; do curl -s http://prod.the-good-parts.com; done | sort
| uniq -c
   9 Hello HTTP World from ip-10-0-116-176.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
  11 Hello HTTP World from ip-10-0-30-144.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
```

terminal

```
$ for run in {1..20}; do curl -s https://prod.the-good-parts.com; done | sort
| uniq -c
  10 Hello HTTPS World from ip-10-0-116-176.ec2.internal in awsbootstrap-
Prod-1PT61TNHUQWTE
  10 Hello HTTPS World from ip-10-0-30-144.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
```

Removing the HTTP endpoint

If this were a production migration, we would first wait for our users to migrate from the HTTP endpoint to HTTPS. Once that happens, we would tear down the HTTP infrastructure.

Let's start by removing all the resources and outputs for the HTTP traffic and endpoints. We have to remove the endpoints before we modify the application code. To do otherwise would lead to failed health checks.

Let's remove the following from `stage.yml`:

- from `Resources`
  - from `SecurityGroup`
    - from `SecurityGroupIngress`
      - the section for traffic on port `8080`
      - the section for traffic on port `80`
  - from `ScalingGroup`

        - the `TargetGroupARNs` entry for `LoadBalancerTargetGroup`
    - the entire `LoadBalancerListener` resource
    - the entire `LoadBalancerTargetGroup` resource
- from `Outputs`
    - the `LBEndpoint` output

And let's remove the following from `main.yml`:

- from `Outputs`
    - the `StagingLBEndpoint` output
    - the `ProdLBEndpoint` output

We can now deploy the updates.

terminal

```
=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Packaging main.yml ===========


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "https://prod.the-good-parts.com",
    "https://staging.the-good-parts.com"
]
```

terminal

```
$ for run in {1..20}; do curl -s https://staging.the-good-parts.com; done |
sort | uniq -c
   9 Hello HTTPS World from ip-10-0-35-100.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
  11 Hello HTTPS World from ip-10-0-64-92.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
```

terminal

```
$ for run in {1..20}; do curl -s https://prod.the-good-parts.com; done | sort
| uniq -c
   9 Hello HTTPS World from ip-10-0-21-46.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
  11 Hello HTTPS World from ip-10-0-68-190.ec2.internal in awsbootstrap-Prod-
1PT61TNHUQWTE
```

Now, our HTTP endpoints don't exist anymore, and hitting them will result in a time out.

terminal

```
$ curl -v http://prod.the-good-parts.com
* Rebuilt URL to: http://prod.the-good-parts.com/
*   Trying 35.153.128.232...
* TCP_NODELAY set
* Connection failed
* connect to 35.153.128.232 port 80 failed: Operation timed out
*   Trying 54.147.46.5...
* TCP_NODELAY set
* Connection failed
* connect to 54.147.46.5 port 80 failed: Operation timed out
* Failed to connect to prod.the-good-parts.com port 80: Operation timed out
* Closing connection 0
curl: (7) Failed to connect to prod.the-good-parts.com port 80: Operation
timed out
```

Let's commit our infrastructure changes.

terminal

```
$ git add main.yml stage.yml
$ git commit -m "Remove HTTP endpoint"
$ git push
```

And now, all that remains is to update our application to stop listening for HTTP requests.

server.js

```
const { hostname } = require('os');
const https = require('https');
const fs = require('fs');

const STACK_NAME = process.env.STACK_NAME || "Unknown Stack";
const httpsPort = 8443;
const httpsKey = '../keys/key.pem'
const httpsCert = '../keys/cert.pem'

if (fs.existsSync(httpsKey) && fs.existsSync(httpsCert)) {
  console.log('Starting https server')
  const message = `Hello HTTPS World from ${hostname()} in ${STACK_NAME}\n`;
  const options = { key: fs.readFileSync(httpsKey), cert:
fs.readFileSync(httpsCert) };
  const server = https.createServer(options, (req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end(message);
  });
  server.listen(httpsPort, hostname, () => {
    console.log(`Server running at http://${hostname()}:${httpsPort}/`);
  });
} else {
  console.log('Could not find certificate/key');
}
```

Let's push our application change to GitHub and wait for it to go through the pipeline.

terminal

```
$ git add server.js
$ git commit -m "Remove http listener from the application"
$ git push
```

We've successfully migrated our application from HTTP only to HTTPS only. We did this without affecting users, by being thoughtful about the phases of our migration. Next, we will improve the network security of our stack.

Network Security

Objective

Make our instances inaccessible from the internet.

Steps

1. Add private subnets with a NAT gateway.
2. Switch our ASGs to use the private subnets.
3. Only allow the HTTPS port in the public subnets.

In this section, we're going to make our EC2 instances inaccessible from the internet. The instances will be able to reach the internet using a NAT gateway, but the network will not allow anything from the internet to reach the instances without going through the load balancer.



Once we make our instances use a NAT gateway to connect to the internet, an additional data transfer charge (currently $0.045/GB in us-east-1) will apply to all traffic that transits the gateway. This includes traffic to other AWS services.

To avoid the extra charge, most AWS services can be configured to expose an endpoint that doesn't pass through the internet. This can be done via Gateway VPC Endpoints or Interface VPC Endpoints (AWS PrivateLink).

Set up SSM for SSH access

One thing that won't work after we lock down our hosts is EC2 Instance Connect. In order to acquire SSH access to our instances we'll have to use AWS Systems Manager Session Manager (SSM).

Let's start by adding two new managed policies to the IAM role used by our instances.

stage.yml

```
InstanceRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        Effect: Allow
        Principal:
          Service:
            - "ec2.amazonaws.com"
        Action: sts:AssumeRole
    ManagedPolicyArns:
      - arn:aws:iam::aws:policy/CloudWatchFullAccess
      - arn:aws:iam::aws:policy/service-role/AmazonEC2RoleforAWSCodeDeploy
      - arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore (1)
      - arn:aws:iam::aws:policy/CloudWatchAgentServerPolicy (1)
    Policies:
      - PolicyName: ec2DescribeTags
        PolicyDocument:
          Version: 2012-10-17
          Statement:
            - Effect: Allow
              Action: 'ec2:DescribeTags'
              Resource: '*'
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

(1) New policies required to allow SSM access.

Let's check it in, and deploy.

terminal

```
$ git add stage.yml
$ git commit -m "Add SSM policies to hosts"
$ git push
```

terminal

```
$ ./deploy-infra.sh
  ...
```

To connect from our local terminal, we can install the

SSM plugin for the AWS CLI while the CloudFormation changes are deploying.

terminal

```
$ curl "https://s3.amazonaws.com/session-manager-
downloads/plugin/latest/mac/sessionmanager-bundle.zip" -o "sessionmanager-
bundle.zip"
  ...
$ unzip sessionmanager-bundle.zip
  ...
$ sudo ./sessionmanager-bundle/install -i /usr/local/sessionmanagerplugin -b
/usr/local/bin/session-manager-plugin  ...
```



Sometimes it can take a while for the updated IAM role to take effect on instances that are already running. If your instances are failing to connect via SSM, you can try terminating your instances and letting the ASG replace them with fresh ones.

We should now be able to open a shell on a remote host with the AWS CLI. Once the connection goes through, we are connected as `ssm-user`, not our familiar `ec2-user`. So before we do anything else, we must switch to `ec2-user` and change into the appropriate home directory.

.terminal

```
$ aws ssm start-session --profile awsbootstrap --target i-07c5a5b5907d43ca7
❶

Starting session with SessionId: josh-0024a17ad7747b5e6
sh-4.2$ sudo su ec2-user
[ec2-user@ip-10-0-192-31 bin]$ pwd
/usr/bin
[ec2-user@ip-10-0-192-31 bin]$ cd ~
[ec2-user@ip-10-0-192-31 ~]$ pwd
/home/ec2-user

[ec2-user@ip-10-0-52-140 ~]$ # admin commands here

[ec2-user@ip-10-0-52-140 ~]$ exit
exit
sh-4.2$ exit
exit


Exiting session with sessionId: josh-0024a17ad7747b5e6.
```

❶ Replace `i-07c5a5b5907d43ca7` with an instance ID from your fleet.

We can also continue to use the AWS console to SSH to our instances. But now we have to choose *Session Manager* instead of *EC2 Instance Connect*.

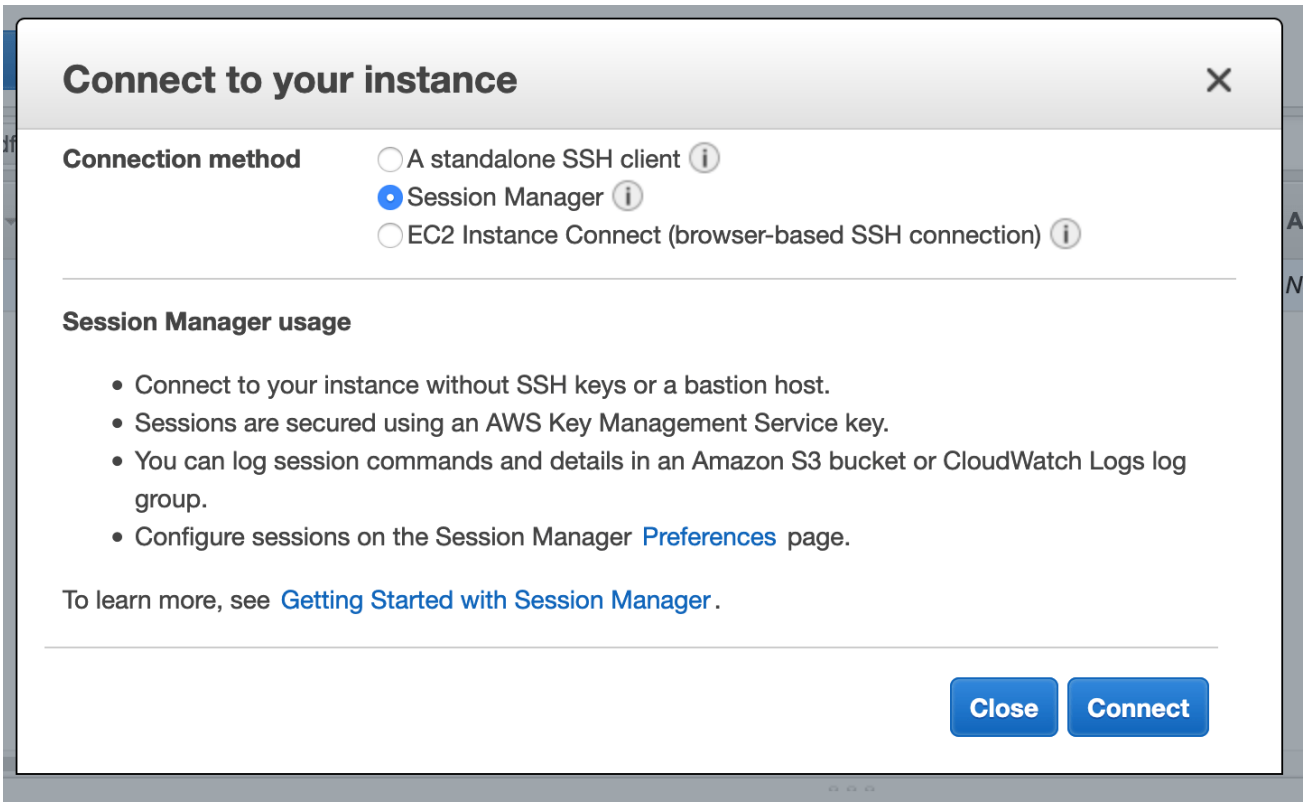


Figure 24. SSM Connection

Add private subnets and NAT gateway

Now, we're going to add new security groups for our private subnets that allows ports 22 and 8443 only.

stage.yml

```
PrivateSecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId: !Ref VPC
    GroupDescription:
      !Sub 'Internal Security group for ${AWS::StackName}'
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 8443
        ToPort: 8443
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: 0.0.0.0/0
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

Next, we have to change the `SecurityGroupIds` property inside the `InstanceLaunchTemplate` resource, so that it refers to `PrivateSecurityGroup.GroupId` instead of `SecurityGroup.GroupId`. In this way, new instances automatically become part of our new private security group.

stage.yml

```
SecurityGroupIds:
  - !GetAtt PrivateSecurityGroup.GroupId
```

Next, we add a new subnet per availability zone with `MapPublicIpOnLaunch` set to `false`, and a CIDR range that doesn't overlap with any of our other subnets.

stage.yml

```
PrivateSubnetAZ1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref VPC
    AvailabilityZone: !Select [ 0, !GetAZs '' ]
    CidrBlock: 10.0.128.0/18
    MapPublicIpOnLaunch: false
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 0, !GetAZs '' ]

PrivateSubnetAZ2:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref VPC
    AvailabilityZone: !Select [ 1, !GetAZs '' ]
    CidrBlock: 10.0.192.0/18
    MapPublicIpOnLaunch: false
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 1, !GetAZs '' ]
```

Now we must create an Elastic IP address for each NAT gateway.

stage.yml

```
EIPAZ1:
  Type: AWS::EC2::EIP
  DependsOn: InternetGatewayAttachment
  Properties:
    Domain: vpc

EIPAZ2:
  Type: AWS::EC2::EIP
  DependsOn: InternetGatewayAttachment
  Properties:
    Domain: vpc
```

Next, let's add the NAT gateways.

stage.yml

```
NATGatewayAZ1:
  Type: AWS::EC2::NatGateway
  Properties:
    AllocationId: !GetAtt EIPAZ1.AllocationId
    SubnetId: !Ref SubnetAZ1
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 0, !GetAZs '' ]

NATGatewayAZ2:
  Type: AWS::EC2::NatGateway
  Properties:
    AllocationId: !GetAtt EIPAZ2.AllocationId
    SubnetId: !Ref SubnetAZ2
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 1, !GetAZs '' ]
```

Now let's add route tables to map outgoing internet traffic to the NAT gateways.

stage.yml

```
PrivateSubnetRouteTableAZ1:
  Type: AWS::EC2::RouteTable
  Properties:
    VpcId: !Ref VPC
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 0, !GetAZs '' ]

PrivateSubnetRouteTableAZ2:
  Type: AWS::EC2::RouteTable
  Properties:
    VpcId: !Ref VPC
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
      - Key: AZ
        Value: !Select [ 1, !GetAZs '' ]

PrivateRouteAZ1:
  Type: AWS::EC2::Route
  Properties:
    RouteTableId: !Ref PrivateSubnetRouteTableAZ1
    DestinationCidrBlock: 0.0.0.0/0
    NatGatewayId: !Ref NATGatewayAZ1

PrivateRouteAZ2:
  Type: AWS::EC2::Route
  Properties:
    RouteTableId: !Ref PrivateSubnetRouteTableAZ2
    DestinationCidrBlock: 0.0.0.0/0
    NatGatewayId: !Ref NATGatewayAZ2

PrivateSubnetRouteTableAssociationAZ1:
  Type: AWS::EC2::SubnetRouteTableAssociation
  Properties:
    RouteTableId: !Ref PrivateSubnetRouteTableAZ1
    SubnetId: !Ref PrivateSubnetAZ1

PrivateSubnetRouteTableAssociationAZ2:
  Type: AWS::EC2::SubnetRouteTableAssociation
  Properties:
    RouteTableId: !Ref PrivateSubnetRouteTableAZ2
    SubnetId: !Ref PrivateSubnetAZ2
```

Switching our ASG to use private subnets

Finally, we have to switch the ASG to launch new instances in the private subnets rather than the public. The instances in the public subnets won't be terminated until the new ones in the private subnets are launched.

Let's change the `VPCZoneIdentifier` in `ScalingGroup` to refer to `PrivateSubnetAZ1` and `PrivateSubnetAZ2` instead of `SubnetAZ1` and `SubnetAZ2`.

stage.yml

```
VPCZoneIdentifier:
  - !Ref PrivateSubnetAZ1
  - !Ref PrivateSubnetAZ2
```

Now we can deploy our infrastructure changes.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Packaging main.yml ===========


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "https://prod.the-good-parts.com",
    "https://staging.the-good-parts.com"
]
```

After the new instances have been launched in the new private subnets, and the old ones have been terminated, we can verify that our application is still reachable through the load balancer endpoints.

terminal

```
$ for run in {1..20}; do curl -s https://staging.the-good-parts.com; done |
sort | uniq -c
  10 Hello HTTPS World from ip-10-0-187-72.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
  10 Hello HTTPS World from ip-10-0-222-16.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
```

terminal

```
$ for run in {1..20}; do curl -s https://prod.the-good-parts.com; done | sort
| uniq -c
  10 Hello HTTPS World from ip-10-0-128-220.ec2.internal in awsbootstrap-
Prod-1PT61TNHUQWTE
  10 Hello HTTPS World from ip-10-0-248-112.ec2.internal in awsbootstrap-
Prod-1PT61TNHUQWTE
```

And now is a good time to push all our changes to GitHub.

terminal

```
$ git add stage.yml
$ git commit -m "Move instances into private subnets"
$ git push
```

Allow only the HTTPS port in public subnets

Once the hosts are running inside private subnets and with the private security group, we can remove ports 8443 and 22 from the public security group. If we had done this in the previous step, it would have prevented users from reaching our application until the new hosts were created.

stage.yml

```
SecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId: !Ref VPC
    GroupDescription:
      !Sub 'Security group for ${AWS::StackName}'
    SecurityGroupIngress: ❶
      - IpProtocol: tcp
        FromPort: 443
        ToPort: 443
        CidrIp: 0.0.0.0/0
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

❶ Only port 443 is allowed in the public subnet.

Now let's deploy and test.

terminal

```
$ ./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Packaging main.yml ===========


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "https://prod.the-good-parts.com",
    "https://staging.the-good-parts.com"
]
```

terminal

```
$ for run in {1..20}; do curl -s https://staging.the-good-parts.com; done |
sort | uniq -c
  11 Hello HTTPS World from ip-10-0-187-72.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
  9 Hello HTTPS World from ip-10-0-222-16.ec2.internal in awsbootstrap-
Staging-10LP6MF0TQC9Y
```

terminal

```
$ for run in {1..20}; do curl -s https://prod.the-good-parts.com; done | sort
| uniq -c
  10 Hello HTTPS World from ip-10-0-128-220.ec2.internal in awsbootstrap-
Prod-1PT61TNHUQWTE
  10 Hello HTTPS World from ip-10-0-248-112.ec2.internal in awsbootstrap-
Prod-1PT61TNHUQWTE
```

Our instances are now isolated from the internet, and the only way to reach them is through the load balancer.

terminal

```
$ git add stage.yml
$ git commit -m "Only allow port 443 in public subnet"
$ git push
```



If you've followed along for the purposes of experimentation, now is a good time to delete all resources so that you won't continue to be charged. Delete the two root CloudFormation stacks, `awsbootstrap` and `awsbootstrap-setup`. Since CloudFormation won't delete non-empty S3 buckets, you'll have to separately delete the buckets that we created to hold our CodePipeline artifacts and CloudFormation templates.